

INTRODUCTION À L'ALGORITHMIQUE

Cours et exercices

Thomas Cormen

Professeur associé d'informatique au Dartmouth College

Charles Leiserson

Professeur d'informatique au MIT

Ronald Rivest

Professeur d'informatique au MIT

Clifford Stein

Professeur associé au génie industriel
et de recherche opérationnelle à l'université de Columbia

Préface de

Philippe chrétienne , Claire Hanen, Alix Munier, Christophe Picouleau

1^{ère} édition traduite de l'américain par Xavier Cazin

Compléments et mises à jour de la 2^e édition traduits par Georges-Louis Kocher

2^e édition

DUNOD

L'édition originale de ce livre a été publiée aux États-Unis par The MIT Press, Cambridge, Massachusetts, sous le titre *Introduction to Algorithms*, second edition.

© The Massachusetts Institute of Technology, 2001
First edition 1990

Ce pictogramme mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du **photocopillage**.

Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les

établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.



Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation du Centre français d'exploitation du droit de copie (**CFC**, 20 rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 1994, pour la 1^{ère} édition
© Dunod, Paris, 2004, pour la présente édition
ISBN 2 10 003922 9

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal. • Seules sont autorisées (Art L 122-5) les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d'information de l'œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

Table des matières

PRÉFACE À L'ÉDITION FRANÇAISE	XVII
PRÉFACE	XXI
PARTIE 1 • INTRODUCTION	
CHAPITRE 1 • RÔLE DES ALGORITHMES EN INFORMATIQUE	3
1.1 Algorithmes	3
Exercices	8
1.2 Algorithmes en tant que technologie	8
Exercices	11
PROBLÈMES	11
CHAPITRE 2 • PREMIERS PAS	13
2.1 Tri par insertion	13
Exercices	18
2.2 Analyse des algorithmes	19
Exercices	25
2.3 Conception des algorithmes	25
Exercices	34
PROBLÈMES	35
CHAPITRE 3 • CROISSANCE DES FONCTIONS	39
3.1 Notation asymptotique	40
Exercices	48
3.2 Notations standard et fonctions classiques	48
Exercices	54
PROBLÈMES	55

CHAPITRE 4 • RÉCURRENCES	59
4.1 Méthode de substitution	60
Exercices	64
4.2 Méthode de l'arbre récursif	64
Exercices	68
4.3 Méthode générale	69
Exercices	71
4.4 Démonstration du théorème général	72
Exercices	80
PROBLÈMES	80
CHAPITRE 5 • ANALYSE PROBABILISTE ET ALGORITHMES RANDOMISÉS	87
5.1 Le problème de l'embauche	87
Exercices	90
5.2 Variables indicatrices	91
Exercices	94
5.3 Algorithmes randomisés	95
Exercices	100
5.4 Analyse probabiliste et autres emplois des variables indicatrices	101
Exercices	112
PROBLÈMES	113
PARTIE 2 • TRI ET RANGS	
CHAPITRE 6 • TRI PAR TAS	121
6.1 Tas	121
Exercices	123
6.2 Conservation de la structure de tas	124
Exercices	125
6.3 Construction d'un tas	126
Exercices	128
6.4 Algorithme du tri par tas	129
Exercices	129
6.5 Files de priorité	131
Exercices	134
PROBLÈMES	135

CHAPITRE 7 • TRI RAPIDE	139
7.1 Description du tri rapide	139
Exercices	142
7.2 Performances du tri rapide	143
Exercices	146
7.3 Versions randomisées du tri rapide	147
Exercices	148
7.4 Analyse du tri rapide	148
Exercices	152
PROBLÈMES	153
CHAPITRE 8 • TRI EN TEMPS LINÉAIRE	159
8.1 Minorants pour le tri	159
Exercices	161
8.2 Tri par dénombrement	162
Exercices	164
8.3 Tri par base	164
Exercices	167
8.4 Tri par paquets	167
Exercices	171
PROBLÈMES	171
CHAPITRE 9 • MÉDIANS ET RANGS	177
9.1 Minimum et maximum	178
Exercices	179
9.2 Sélection en temps moyen linéaire	179
Exercices	183
9.3 Sélection en temps linéaire dans le cas le plus défavorable	183
Exercices	186
PROBLÈMES	187
PARTIE 3 • STRUCTURES DE DONNÉES	
CHAPITRE 10 • STRUCTURES DE DONNÉES ÉLÉMENTAIRES	195
10.1 Piles et files	195
Exercices	197

10.2 Listes chaînées	199
Exercices	203
10.3 Implémentation des pointeurs et des objets	203
Exercices	207
10.4 Représentation des arborescences	208
Exercices	209
PROBLÈMES	211
CHAPITRE 11 • TABLES DE HACHAGE	215
11.1 Tables à adressage direct	216
Exercices	217
11.2 Tables de hachage	218
Exercices	222
11.3 Fonctions de hachage	223
Exercices	230
11.4 Adressage ouvert	231
Exercices	238
11.5 Hachage parfait	238
Exercices	242
PROBLÈMES	243
CHAPITRE 12 • ARBRES BINAIRES DE RECHERCHE	247
12.1 Qu'est-ce qu'un arbre binaire de recherche ?	248
Exercices	249
12.2 Requête dans un arbre binaire de recherche	250
Exercices	253
12.3 Insertion et suppression	254
Exercices	257
12.4 Arbres binaires de recherche construits aléatoirement	258
Exercices	261
PROBLÈMES	262
CHAPITRE 13 • ARBRES ROUGE-NOIR	267
13.1 Propriétés des arbres rouge-noir	267
Exercices	270
13.2 Rotation	271
Exercices	272

13.3 Insertion	273
Exercices	280
13.4 Suppression	281
Exercices	286
PROBLÈMES	287
CHAPITRE 14 • EXTENSION D'UNE STRUCTURE DE DONNÉES	295
14.1 Rangs dynamiques	296
Exercices	300
14.2 Comment étendre une structure de données	301
Exercices	303
14.3 Arbres d'intervalles	304
Exercices	309
PROBLÈMES	310
PARTIE 4 • TECHNIQUES AVANCÉES DE CONCEPTION ET D'ANALYSE	
CHAPITRE 15 • PROGRAMMATION DYNAMIQUE	315
15.1 Ordonnement de chaînes de montage	316
Exercices	322
15.2 Multiplications matricielles enchaînées	323
Exercices	330
15.3 Éléments de la programmation dynamique	330
Exercices	341
15.4 Plus longue sous-séquence commune	341
Exercices	347
15.5 Arbres binaires de recherche optimaux	347
Exercices	354
PROBLÈMES	354
CHAPITRE 16 • ALGORITHMES GLOUTONS	361
16.1 Un problème de choix d'activités	362
Exercices	370
16.2 Éléments de la stratégie gloutonne	370
Exercices	375
16.3 Codages de Huffman	376
Exercices	382

16.4 Fondements théoriques des méthodes gloutonnes	383
Exercices	388
16.5 Un problème d'ordonnement de tâches	389
Exercices	392
PROBLÈMES	392
CHAPITRE 17 • ANALYSE AMORTIE	395
17.1 Méthode de l'agrégat	396
Exercices	400
17.2 Méthode comptable	400
Exercices	402
17.3 Méthode du potentiel	402
Exercices	405
17.4 Tables dynamiques	406
Exercices	414
PROBLÈMES	415
PARTIE 5 • STRUCTURES DE DONNÉES AVANCÉES	
CHAPITRE 18 • B-ARBRES	425
18.1 Définition d'un B-arbre	429
Exercices	431
18.2 Opérations fondamentales sur les B-arbres	432
Exercices	437
18.3 Suppression d'une clé dans un B-arbre	439
Exercices	442
PROBLÈMES	442
CHAPITRE 19 • TAS BINOMIAUX	445
19.1 Arbres binomiaux et tas binomiaux	447
Exercices	450
19.2 Opérations sur les tas binomiaux	451
Exercices	461
PROBLÈMES	462

CHAPITRE 20 • TAS DE FIBONACCI	465
20.1 Structure des tas de Fibonacci	466
20.2 Opérations sur les tas fusionnables	469
Exercices	477
20.3 Diminution d'une clé et suppression d'un nœud	478
Exercices	481
20.4 Borne pour le degré maximal	482
Exercices	484
PROBLÈMES	484
CHAPITRE 21 • STRUCTURES DE DONNÉES POUR ENSEMBLES DISJOINTS	487
21.1 Opérations sur les ensembles disjoints	487
Exercices	490
21.2 Représentation d'ensembles disjoints par des listes chaînées	490
Exercices	493
21.3 Forêts d'ensembles disjoints	494
Exercices	497
21.4 Analyse de l'union par rang avec compression de chemin	498
Exercices	505
PROBLÈMES	506
PARTIE 6 • ALGORITHMES POUR LES GRAPHES	
CHAPITRE 22 • ALGORITHMES ÉLÉMENTAIRES POUR LES GRAPHES	513
22.1 Représentation des graphes	514
Exercices	516
22.2 Parcours en largeur	517
Exercices	524
22.3 Parcours en profondeur	525
Exercices	532
22.4 Tri topologique	534
Exercices	536
22.5 Composantes fortement connexes	536
Exercices	541
PROBLÈMES	542

CHAPITRE 23 • ARBRES COUVRANTS DE POIDS MINIMUM	545
23.1 Construction d'un arbre couvrant minimum	546
Exercices	550
23.2 Algorithmes de Kruskal et de Prim	551
Exercices	556
PROBLÈMES	558
CHAPITRE 24 • PLUS COURTS CHEMINS À ORIGINE UNIQUE	563
24.1 Algorithme de Bellman-Ford	571
Exercices	574
24.2 Plus courts chemins à origine unique dans les graphes orientés sans circuit	575
Exercices	577
24.3 Algorithme de Dijkstra	577
Exercices	582
24.4 Contraintes de potentiel et plus courts chemins	583
Exercices	587
24.5 Démonstrations des propriétés de plus court chemin	589
Exercices	594
PROBLÈMES	595
CHAPITRE 25 • PLUS COURTS CHEMINS POUR TOUT COUPLE DE SOMMETS	601
25.1 Plus courts chemins et multiplication de matrices	603
Exercices	608
25.2 L'algorithme de Floyd-Warshall	609
Exercices	614
25.3 Algorithme de Johnson pour les graphes peu denses	616
Exercices	620
PROBLÈMES	621
CHAPITRE 26 • FLOT MAXIMUM	625
26.1 Réseaux de transport	626
Exercices	631
26.2 La méthode de Ford-Fulkerson	632
Exercices	643
26.3 Couplage maximum dans un graphe biparti	644
Exercices	648
26.4 Algorithmes de préflots	649
Exercices	658

26.5 Algorithme réétiqueter-vers-l'avant	659
Exercices	669
PROBLÈMES	669

PARTIE 7 • MORCEAUX CHOISIS

CHAPITRE 27 • RÉSEAUX DE TRI	681
27.1 Réseaux de comparaison	682
Exercices	685
27.2 Le principe du zéro-un	686
Exercices	688
27.3 Un réseau de tri bitonique	689
Exercices	690
27.4 Un réseau de fusion	692
Exercices	693
27.5 Un réseau de tri	694
Exercices	696
PROBLÈMES	697
CHAPITRE 28 • CALCUL MATRICIEL	701
28.1 Propriétés des matrices	702
Exercices	709
28.2 Algorithme de Strassen pour la multiplication des matrices	710
Exercices	716
28.3 Résolution de systèmes d'équations linéaires	717
Exercices	730
28.4 Inversion des matrices	730
Exercices	734
28.5 Matrices symétriques définies positives et approximation des moindres carrés	735
Exercices	740
PROBLÈMES	741
CHAPITRE 29 • PROGRAMMATION LINÉAIRE	745
29.1 Forme canonique et forme standard	752
Exercices	759
29.2 Formulation de problèmes comme programmes linéaires	760
Exercices	764

29.3	Algorithme du simplexe	765
	Exercices	778
29.4	Dualité	779
	Exercices	784
29.5	Solution de base réalisable initiale	785
	Exercices	790
	PROBLÈMES	791
CHAPITRE 30 • POLYNÔMES ET TRANSFORMÉE RAPIDE DE FOURIER		795
30.1	Représentation des polynômes	797
	Exercices	802
30.2	Transformée discrète de Fourier et transformée rapide de Fourier	803
	Exercices	810
30.3	Implémentations efficaces de la FFT	811
	Exercices	816
	PROBLÈMES	816
CHAPITRE 31 • ALGORITHMES DE LA THÉORIE DES NOMBRES		821
31.1	Notions de théorie des nombres	823
	Exercices	827
31.2	Plus grand commun diviseur	828
	Exercices	832
31.3	Arithmétique modulaire	833
	Exercices	839
31.4	Résolution d'équations linéaires modulaires	839
	Exercices	842
31.5	Théorème du reste chinois	843
	Exercices	845
31.6	Puissances d'un élément	846
	Exercices	850
31.7	Le cryptosystème à clés publiques RSA	850
	Exercices	856
31.8	Test de primalité	856
	Exercices	865
31.9	Factorisation des entiers	865
	Exercices	870
	PROBLÈMES	870

CHAPITRE 32 • RECHERCHE DE CHAÎNES DE CARACTÈRES	875
32.1 Algorithme naïf de recherche de chaîne de caractères	878
Exercices	879
32.2 Algorithme de Rabin-Karp	880
Exercices	884
32.3 Recherche de chaîne de caractères au moyen d'automates finis	885
Exercices	891
32.4 Algorithme de Knuth-Morris-Pratt	891
Exercices	898
PROBLÈMES	899
CHAPITRE 33 • GÉOMÉTRIE ALGORITHMIQUE	901
33.1 Propriétés des segments de droite	902
Exercices	907
33.2 Déterminer si deux segments donnés se coupent	908
Exercices	914
33.3 Recherche de l'enveloppe convexe	915
Exercices	924
33.4 Recherche des deux points les plus rapprochés	925
Exercices	929
PROBLÈMES	930
CHAPITRE 34 • NP-COMPLÉTUDE	933
34.1 Temps polynomial	939
Exercices	945
34.2 Vérification en temps polynomial	946
Exercices	950
34.3 NP-complétude et réductibilité	951
Exercices	960
34.4 Preuves de NP-complétude	961
Exercices	968
34.5 Problèmes NP-complets	969
Exercices	982
PROBLÈMES	983

CHAPITRE 35 • ALGORITHMES D'APPROXIMATION	987
35.1 Problème de la couverture de sommets	989
Exercices	992
35.2 Problème du voyageur de commerce	992
Exercices	997
35.3 Problème de la couverture d'ensemble	997
Exercices	1002
35.4 Randomisation et programmation linéaire	1002
Exercices	1007
35.5 Problème de la somme de sous-ensemble	1007
Exercices	1012
PROBLÈMES	1013

PARTIE 8 • ANNEXES : ÉLÉMENTS DE MATHÉMATIQUES

ANNEXE A • SOMMATIONS	1021
A.1 Formules et propriétés des sommations	1022
Exercices	1025
A.2 Bornes des sommations	1025
Exercices	1031
PROBLÈMES	1031
ANNEXE B • ENSEMBLES, ETC.	1033
B.1 Ensembles	1033
Exercices	1037
B.2 Relations	1038
Exercices	1040
B.3 Fonctions	1040
Exercices	1042
B.4 Graphes	1043
Exercices	1046
B.5 Arbres	1047
Exercices	1053
PROBLÈMES	1054

ANNEXE C • DÉNOMBREMENT ET PROBABILITÉS	1057
C.1 Dénombrement	1057
Exercices	1061
C.2 Probabilités	1063
Exercices	1068
C.3 Variables aléatoires discrètes	1069
Exercices	1073
C.4 Distributions géométrique et binomiale	1074
Exercices	1078
C.5 Queues de la distribution binomiale	1079
Exercices	1084
PROBLÈMES	1085
BIBLIOGRAPHIE	1087
INDEX	1109

Préface à l'édition française

Vous savez compter. Un ordinateur aussi ! Mais connaissez-vous les mécanismes utilisés ? Etes-vous vraiment sûr que le résultat affiché soit juste ? Combien de temps devrez-vous attendre la fin du calcul ? N'y a-t-il pas un moyen de l'obtenir plus vite ? Que vous soyez ingénieur, mathématicien, physicien, statisticien et surtout informaticien, toutes ces questions vous vous les posez. Si vous êtes étudiant, elles surgiront très rapidement.

Étudier l'algorithmique, c'est apporter des réponses à vos questions.

Cette science est le cœur de l'informatique. Pour tout ceux qui doivent ou devront faire travailler un ordinateur, il est essentiel de comprendre ses principes fondamentaux et de connaître ses éléments de base. Une formule 1 ne se conduit pas comme une voiture à pédales. De même un ordinateur se s'utilise pas comme un boulier. L'algorithmique est le permis de conduire de l'informatique. Sans elle, il n'est pas concevable d'exploiter sans risque un ordinateur.

Cette introduction remarquable à l'algorithmique donne au lecteur d'une part les bases théoriques indispensables et lui fournit d'autre part les moyens de concevoir rigoureusement des programmes efficaces permettant de résoudre des problèmes variés issus de différentes applications.

L'éventail des algorithmes présentés va des plus classiques, comme les algorithmes de tri et les fonctions de hachage, aux plus récents comme ceux de la cryptographie. On trouve ici rassemblés des algorithmes numériques, par exemple pour l'inversion de matrices ou la transformée de Fourier et des algorithmes combinatoires comme les algorithmes de graphes ou la recherche de motif.

Une très large place est faite aux structures de données, des plus simples comme les listes, aux plus sophistiquées comme les tas de Fibonacci. Notons au passage l'importance accordée aux différentes mesures de complexité (pire des cas, amortissement, en moyenne) qui permettent d'approfondir entre autres l'étude de l'efficacité des algorithmes de tri et des structures de données.

Il est certain que la plupart des informaticiens spécialisés trouveront dans ce livre leurs algorithmes de base, exprimés de façon unifiée, ainsi que certaines avancées récentes dans leur domaine. Cet ouvrage met donc en relief le rôle central joué par l'algorithmique dans la science Informatique.

La présence de chapitres méthodologiques comme ceux consacrés à la programmation dynamique et aux algorithmes gloutons, ainsi que les deux derniers qui traitent de la complexité des problèmes et de la conception d'algorithmes approchés, permet au lecteur d'amorcer une réflexion plus poussée sur la manière d'aborder un problème et de concevoir une méthode de résolution. Ces chapitres sont illustrés par des exemples d'application bien choisis et là encore très divers.

Cette gamme de sujets, riche par sa variété et ses niveaux de difficulté, est soutenue par une pédagogie constante, qui rend la lecture de l'ouvrage facile et agréable, sans nuire à l'exigence de rigueur. A titre d'exemple, on peut citer la clarté remarquable des chapitres consacrés aux graphes, qui, à partir d'un algorithme générique, introduisent toute une famille de variantes efficaces dont les différentes implémentations sont analysées avec finesse.

D'une manière générale, les notions présentées sont systématiquement introduites de façon informelle à partir d'un exemple ou d'une application particulière, avant d'être formalisées. Les propriétés et les algorithmes sont toujours démontrés. La présence d'une partie consacrée aux fondements mathématiques utilisés est tout à fait bienvenue et rend l'ouvrage accessible avec très peu de prérequis.

Le lecteur peut facilement se familiariser et approfondir les notions rencontrées grâce aux nombreux exercices de difficulté graduée. Les problèmes permettent d'aller plus loin dans la compréhension du chapitre, et sont souvent une occasion de connaître différentes applications pratiques des algorithmes présentés. De ce point de vue, ce livre est une mine d'or pour tout enseignant d'algorithmique.

La lecture de cet ouvrage est tout à fait recommandée aux étudiants de second et de troisième cycle d'informatique et de mathématiques, ainsi qu'aux élèves ingénieurs. Tous y trouveront une aide et un support de cours utile tout au long de leurs études.

La diversité des sujets abordés, l'efficacité des algorithmes présentés, et leur écriture dans un pseudo-code proche des langages C et Pascal, qui les rend très faciles à implémenter, font aussi de ce livre un recueil fort utile dans la vie professionnelle d'un informaticien ou d'un ingénieur.

Enfin, au delà de ses besoins propres, nous souhaitons que le lecteur, qu'il soit ingénieur, étudiant, ou simplement curieux, prenne comme nous plaisir et intérêt à la lecture de cet ouvrage.

Paris, mars 1994

PHILIPPE CHRÉTIENNE, CLAIRE HANEN,
ALIX MUNIER, CHRISTOPHE PICOULEAU
Université Pierre et Marie Curie (LIP6)

À propos de la seconde édition :

C'est avec une curiosité renouvelée que l'enseignant, le chercheur ou l'étudiant abordera cette seconde édition du livre de référence de l'algorithmique. L'algorithmicien déjà familier de la précédente édition y trouvera, parmi moult enrichissements disséminés tout le long de l'ouvrage, de nouvelles parties, notamment celle dédiée à la programmation linéaire, de nombreux exercices et problèmes inédits, ainsi qu'une présentation uniformisée des preuves d'algorithmes ; le lecteur, étudiant ou enseignant, qui découvre cette *Introduction à l'algorithmique*, y trouvera sous un formalisme des plus limpides, le nécessaire, voire un peu plus, de cette discipline qui est l'une des pierres angulaires de l'informatique.

Paris, août 2002

CHRISTOPHE PICOULEAU,
Laboratoire CEDRIC CNAM

Préface

Nous proposons dans ce livre une introduction complète à l'étude contemporaine des algorithmes informatiques. De nombreux algorithmes y sont présentés et étudiés en détail, de façon à rendre leur conception et leur analyse accessibles à tous les niveaux de lecture. Nous avons essayé de maintenir la simplicité des explications, sans sacrifier ni la profondeur de l'étude, ni la rigueur mathématique.

Chaque chapitre présente un algorithme, une technique de conception, un domaine d'application, ou un sujet s'y rapportant. Les algorithmes sont décrits en français et dans un « pseudo-code » conçu pour être lisible par quiconque ayant déjà un peu programmé. Le livre contient plus de 230 figures qui illustrent le fonctionnement des algorithmes. Comme nous mettons l'accent sur *l'efficacité* comme critère de conception, les temps d'exécution de tous nos algorithmes sont soigneusement analysés.

Ce texte est en premier lieu un support du cours d'algorithmique ou de structures de données de deuxième ou troisième cycle universitaire. Il est également bien adapté à la formation personnelle des techniciens professionnels, puisqu'il s'intéresse aux problèmes d'ingénierie ayant trait à la conception d'algorithmes, ainsi qu'à leurs aspects mathématiques.

Dans cette édition, qui est la seconde, nous avons modifié l'ensemble du livre. Les changements vont de la simple refonte de phrases individuelles jusqu'à l'ajout de nouveaux chapitres.

a) Pour l'enseignant

Ce livre se veut à la fois complet et polyvalent. Il se révélera utile pour toute sorte de cours, depuis un cours de structures de données en deuxième cycle jusqu'à un cours

d'algorithmique en troisième cycle. Un cours trimestriel étant beaucoup trop court pour aborder tous les sujets étudiés ici, on peut voir ce livre comme un « buffet garni » où vous pourrez choisir le matériel le mieux adapté aux cours que vous souhaitez enseigner.

Vous trouverez commode d'organiser votre cours autour des chapitres dont vous avez vraiment besoin. Nous avons fait en sorte que les chapitres soient relativement indépendants, de manière à éviter toute subordination inattendue ou superflue d'un chapitre à l'autre. Chaque chapitre commence en présentant des notions simples et se poursuit avec les notions plus difficiles, le découpage en sections créant des points de passage naturels. Dans un cours de deuxième cycle, on pourra ne faire appel qu'aux premières sections d'un chapitre donné ; en troisième cycle, on pourra considérer le chapitre entier.

Nous avons inclus plus de 920 exercices et plus de 140 problèmes. Chaque section se termine par des *exercices* et chaque chapitre se termine par des *problèmes*. Les exercices sont généralement des questions courtes de contrôle des connaissances. Certains servent surtout à tester la compréhension du sujet ; d'autres, plus substantiels, sont plutôt du genre devoir à la maison. Les problèmes sont des études de cas plus élaborées qui introduisent souvent de nouvelles notions ; ils sont composés le plus souvent de plusieurs questions qui guident l'étudiant à travers les étapes nécessaires pour parvenir à une solution.

Les sections et exercices munis d'une astérisque (*) recouvrent des thèmes destinés plutôt aux étudiants de troisième cycle. Un passage étoilé n'est pas forcément plus ardu qu'un passage non étoilé, mais il risque d'exiger des connaissances mathématiques plus pointues. De même, un exercice étoilé risque de demander un niveau théorique plus important ou d'incorporer des subtilités au-dessus de la moyenne.

b) Pour l'étudiant

Nous espérons que ce livre vous fournira une introduction agréable à l'algorithmique. Nous avons essayé de rendre chaque algorithme accessible et intéressant. Pour vous aider lorsque vous rencontrez des algorithmes peu familiers ou difficiles, nous les avons tous décrits étape par étape. Nous expliquons aussi avec soin les notions mathématiques nécessaires pour comprendre l'analyse des algorithmes. Si vous êtes déjà familiarisé avec un sujet, vous constaterez que l'organisation des chapitres vous permet de sauter les sections d'introduction et d'aller rapidement aux concepts plus avancés.

Ceci est un livre volumineux, et votre cours n'en couvrira sans doute qu'une partie. Nous avons pourtant essayé d'en faire un livre qui vous servira aussi bien maintenant comme support de cours, que plus tard dans votre carrière, comme référence mathématique, ou manuel d'ingénierie.

Quels sont les pré-requis pour lire ce livre ?

- Vous devrez avoir une petite expérience de la programmation. En particulier vous devrez comprendre les procédures récursives et les structures de données simples comme les tableaux et les listes chaînées.
- Vous devrez être relativement familiarisé avec les démonstrations mathématiques par récurrence. Certaines parties de ce livre reposent sur des connaissances de calcul élémentaire. Cela dit, les parties 1 et 8 de ce livre vous apprendront toutes les techniques mathématiques dont vous aurez besoin.

c) Pour le professionnel

La grande variété de sujets présents dans ce livre en fait un excellent manuel de référence sur les algorithmes. Chaque chapitre étant relativement indépendant des autres, vous pourrez vous concentrer sur les sujets qui vous intéressent le plus.

La plupart des algorithmes étudiés ont une grande utilité pratique. Nous mettons donc l'accent sur l'implémentation et les autres problèmes d'ingénierie. Nous offrons le plus souvent des alternatives pratiques aux quelques algorithmes qui sont surtout d'intérêt théorique.

Si vous souhaitez implémenter l'un de ces algorithmes, vous n'aurez aucun mal à traduire notre pseudo-code dans votre langage de programmation favori. Le pseudo-code est conçu pour présenter chaque algorithme de façon claire et succincte. Nous ne nous intéressons donc pas à la gestion des erreurs et autres problèmes de génie logiciel qui demandent des hypothèses particulières sur l'environnement de programmation. Nous essayons de présenter chaque algorithme simplement et directement de manière à éviter que leur essence ne soit masquée par les idiosyncrasies d'un langage de programmation particulier.

d) Pour nos collègues

Nous donnons une bibliographie très complète et des références à la littérature courante. Chaque chapitre se termine par une partie « notes de chapitre » qui fournissent des détails et des références historiques. Les notes de chapitre ne fournissent pas, toutefois, une référence complète au vaste champ des algorithmes. Malgré la taille imposante de cet ouvrage, nous avons dû renoncer, faute de place, à inclure nombre d'algorithmes intéressants.

Nonobstant les innombrables supplications émanant d'étudiants, nous avons décidé de ne pas fournir de solutions aux problèmes et exercices ; ainsi, l'étudiant ne succombera pas à la tentation de regarder la solution au lieu d'essayer de la trouver par lui-même.

e) Modifications apportées à la seconde édition

Qu'est-ce qui a changé par rapport à la première édition de ce livre ? Pas grand chose ou beaucoup de choses, selon le point de vue adopté.

Un examen sommaire de la table des matières montre que la plupart des chapitres de la première édition figurent aussi dans la seconde. Nous avons supprimé deux chapitres et une poignée de sections, mais nous avons ajouté trois nouveaux chapitres et quatre nouvelles sections réparties sur d'autres chapitres. Si vous deviez évaluer l'ampleur des modifications d'après la table des matières, vous en concluriez que les changements ont été limités.

En fait, les modifications vont bien au-delà de ce que semble montrer la table des matières. Voici, dans le désordre, un résumé des changements les plus significatifs pour la seconde édition :

- Cliff Stein a rejoint notre équipe d'auteurs.
- Certaines erreurs ont été corrigées. Combien ? Disons, un certain nombre.
- Il y a trois nouveaux chapitres :
 - Le chapitre 1 présente le rôle des algorithmes en informatique.
 - Le chapitre 5 traite de l'analyse probabiliste et des algorithmes randomisés. Comme avec la première édition, ces thèmes reviennent souvent dans cet ouvrage.
 - Le chapitre 29 est consacré à la programmation linéaire.
- Aux chapitres repris de la première édition ont été ajoutées de nouvelles sections, traitant des sujets que voici :
 - hachage parfait (section 11.5),
 - deux applications de la programmation dynamique (sections 15.1 et 15.5), et
 - algorithmes d'approximation utilisant randomisation et programmation linéaire (section 35.4).
- Pour montrer davantage d'algorithmes en début de livre, trois des chapitres consacrés aux théories mathématiques ont été transférés de la partie 1 vers les annexes, à savoir la partie 8.
- Il y a plus de 40 nouveaux problèmes et plus de 185 nouveaux exercices.
- Nous avons rendu explicite l'utilisation des invariants de boucle pour prouver la validité des algorithmes. Notre premier invariant apparaît au chapitre 2, et nous en verrons une trentaine d'autres tout au long de cet ouvrage.
- Nous avons réécrit nombre des analyses probabilistes. En particulier, nous utilisons à une dizaine d'endroits la technique des « variables indicatrices » qui simplifie les analyses probabilistes, surtout quand les variables aléatoires sont dépendantes.
- Nous avons enrichi et actualisé les notes de chapitre et la bibliographie. Celle-ci a augmenté de plus de 50% et nous avons cité nombre de nouveaux résultats algorithmiques qui ont paru postérieurement à la première édition de ce livre.

Nous avons également procédé aux changements suivants :

- Le chapitre consacré à la résolution des récurrences ne contient plus la méthode d’itération. À la place, dans la section 4.2, nous avons « promu » les arbres récursifs de façon à en faire une méthode de plein droit. Nous avons trouvé que tracer des arbres récursifs est moins sujet à erreur que d’itérer des récurrences. Nous signalons, cependant, que les arbres récursifs sont surtout intéressants pour générer des conjectures qui seront ensuite vérifiées via la méthode de substitution.
- La méthode de partitionnement employée pour le tri rapide (*quicksort*) (section 7.1) et pour la sélection en temps moyen linéaire (section 9.2) a changé. Nous utilisons désormais la méthode de Lomuto qui, grâce à des variables indicatrices, simplifie quelque peu l’analyse. La méthode utilisée dans la première édition, due à Hoare, apparaît sous forme de problème au chapitre 7.
- Nous avons modifié la présentation du hachage universel à la section 11.3.3 de façon que cette étude rentre dans le cadre du hachage parfait.
- Il y a une analyse beaucoup plus simple de la hauteur d’un arbre de recherche binaire généré aléatoirement, à la section 12.4.
- La présentation de la programmation dynamique (section 15.3) et celle des algorithmes gloutons (section 16.2) ont été fortement enrichies. L’étude du problème du choix d’activités, exposé au début du chapitre consacré aux algorithmes gloutons, aide à clarifier les relations entre programmation dynamique et algorithmes gloutons.
- Nous avons remplacé la démonstration du temps d’exécution de la structure de données union d’ensembles disjoints, à la section 21.4, par une démonstration qui emploie la méthode du potentiel pour déterminer une borne serrée.
- La démonstration de la validité de l’algorithme de recherche des composantes fortement connexes, à la section 22.5, est plus simple, plus claire et plus directe.
- Le chapitre 24, consacré aux plus courts chemins à origine unique, a été refondu de façon que les démonstrations des propriétés fondamentales soient placées dans une section spécifique. Cette nouvelle organisation nous permet de commencer plus tôt l’étude des algorithmes.
- La section 34.5 contient une présentation enrichie de la NP-complétude, ainsi que de nouvelles démonstrations de la NP-complétude des problèmes du cycle hamiltonien et de la somme d’un sous-ensemble.

Enfin, nous avons revu quasiment toutes les sections pour corriger, simplifier et clarifier explications et démonstrations.

f) Site web

Un autre changement par rapport à la première édition est ce que le livre a maintenant son site web à lui : <http://mitpress.mit.edu/algorithms/>. Vous pouvez utiliser ce site pour signaler des erreurs, obtenir la liste des erreurs connues ou

émettre des suggestions ; n'hésitez pas à nous faire signe. Nous sommes tout particulièrement intéressés par des idées d'exercice et de problème, mais n'oubliez pas d'y joindre les solutions.

Nous regrettons d'être dans l'impossibilité de répondre personnellement à tous les commentaires.

g) Remerciements pour la première édition

De nombreux amis et collègues ont largement contribué à la qualité de ce livre. Nous les remercions tous pour leur aide et leurs critiques constructives.

Le laboratoire d'informatique du MIT nous a offert un environnement de travail idéal. Nos collègues du groupe de théorie du calcul ont été particulièrement compréhensifs et tolérants quant à nos incessantes demandes d'expertise de tel ou tel chapitre. Nous remercions particulièrement Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys et Eva Tardos. Merci à William Ang, Sally Bemus, Ray Hirschfeld et Mark Reinhold pour avoir permis à nos machines (DEC Microvax, Apple Macintosh et Sparcstation Sun) de fonctionner sans heurts et pour avoir recompilé \TeX chaque fois que nous dépassions une limite de temps de compilation. Thinking Machines Corporation a partiellement permis à Charles Leiserson de travailler à ce livre alors qu'il était absent du MIT.

De nombreux collègues ont utilisé les épreuves de ce texte pour leur Cours dans d'autres universités. Ils ont suggéré bon nombre de corrections et de révisions. Nous souhaitons notamment remercier Richard Beigel, Andrew Goldberg, Joan Lucas, Mark Overmars, Alan Sherman et Diane Souvaine.

Parmi les enseignants qui nous assistent pour nos cours, beaucoup ont contribué de manière significative au développement de ce texte. Nous remercions particulièrement Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel et Margaret Tuttle.

Nombreux sont ceux qui nous ont apporté une assistance technique complémentaire mais précieuse. Denise Sergent a passé de nombreuses heures dans les bibliothèques du MIT, à la recherche de références bibliographiques. Maria Sensale, la bibliothécaire de notre salle de lecture fut toujours souriante et serviable. L'accès à la bibliothèque personnelle d'Albert Meyer nous a économisé de nombreuses heures pendant la rédaction des notes de chapitre. Shlomo Kipnis, Bill Niehaus et David Wilson ont validé les anciens exercices, en ont conçu de nouveaux et ont annoté leurs solutions. Marios Papaefthymiou et Gregory Troxel ont contribué à l'index. Au fil des ans, nos secrétaires Inna Radzihovsky, Denise Sergent, Gayle Sherman, et surtout Be Hubbard, ont constamment soutenu ce projet, et nous les en remercions.

Parmi les erreurs relevées dans les premières épreuves, beaucoup l'ont été par nos étudiants. Nous remercions particulièrement Bobby Blumofe, Bonnie Eisenberg,

Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio, et Margaret Tuttle pour leur lecture attentive.

Nos collègues ont également effectué des relectures critiques de certains chapitres, ou donné des informations sur des algorithmes particuliers, et nous leur en sommes reconnaissants. Nous remercions particulièrement Bill Aiello, Alok Aggarwal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Hershel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan et Paul Wang. Plusieurs de nos collègues nous ont aussi fourni gracieusement quelques *problèmes* ; nous remercions notamment Andrew Goldberg, Danny Sleator et Umesh Vazirani.

Nous avons eu plaisir à travailler avec MIT Press et McGraw-Hill pendant la mise en forme de ce texte. Nous remercions particulièrement Frank Satlow, Terry Ehling, Larry Cohen et Lorrie Lejeune de MIT Press et David Shapiro de McGraw-Hill pour leur encouragement, leur soutien et leur patience. Nous sommes particulièrement reconnaissant à Larry Cohen pour son exceptionnelle correction d'épreuves.

h) Remerciements pour la seconde édition

Quand nous demandâmes à Julie Sussman, P.P.A., de nous servir de correctrice technique pour cette seconde édition, nous ne savions pas que nous allions tomber sur l'oiseau rare. En plus de vérifier le contenu technique, Julie a corrigé avec ardeur notre prose. C'est pour nous une belle leçon d'humilité que de voir le nombre d'erreurs que Julie a trouvées dans nos premières épreuves ; encore que, compte tenu du nombre d'erreurs qu'elle avait décelées dans la première édition (après publication, héla), il n'y a rien d'étonnant à cela. Qui plus est, Julie a sacrifié son emploi du temps personnel pour favoriser le nôtre ; elle a même relu des chapitres pendant des vacances aux Iles Vierges ! Julie, comment pourrions-nous vous remercier pour le travail incroyable que vous avez effectué ?

La seconde édition a été préparée alors que les auteurs faisaient partie du département d'informatique de Dartmouth College et du laboratoire d'informatique du MIT. Ces deux environnements ne pouvaient être que stimulants, et nous remercions nos collègues pour leur aide.

Des amis et des collègues, un peu partout dans le monde, nous ont aidé, par leurs suggestions et leurs avis autorisés, à améliorer notre texte. Grand merci à Sanjeev Arora, Javed Aslam, Guy Blelloch, Avrim Blum, Scot Drysdale, Hany Farid, Hal Gabow, Andrew Goldberg, David Johnson, Yanlin Liu, Nicolas Schabanel, Alexander Schrijver, Sasha Shen, David Shmoys, Dan Spielman, Gerald Jay Sussman, Bob Tarjan, Mikkel Thorup et Vijay Vazirani.

De nombreux enseignants et collègues nous ont appris beaucoup de choses sur les algorithmes. Nous remercions tout particulièrement nos professeurs Jon L. Bentley, Bob Floyd, Don Knuth, Harold Kuhn, H. T. Kung, Richard Lipton, Arnold Ross,

Larry Snyder, Michael I. Shamos, David Shmoys, Ken Steiglitz, Tom Szymanski, Éva Tardos, Bob Tarjan et Jeffrey Ullman.

Nous remercions, pour leur contribution, tous les assistants chargés de cours d'algorithmique au MIT et à Dartmouth, dont Joseph Adler, Craig Barrack, Bobby Blumofe, Roberto De Prisco, Matteo Frigo, Igal Galperin, David Gupta, Raj D. Iyer, Nabil Kahale, Sarfraz Khurshid, Stavros Kolliopoulos, Alain Leblanc, Yuan Ma, Maria Minkoff, Dimitris Mitsouras, Alin Popescu, Harald Prokop, Sudipta Sengupta, Donna Slonim, Joshua A. Tauber, Sivan Toledo, Elisheva Werner-Reiss, Lea Wittie, Qiang Wu et Michael Zhang.

L'assistance informatique nous a été fournie par William Ang, Scott Blomquist et Greg Shomo au MIT, et par Wayne Cripps, John Konkle et Tim Tregubov à Dartmouth. Merci également à Be Blackburn, Don Dailey, Leigh Deacon, Irene Sebeda et Cheryl Patton Wu du MIT, et à Phyllis Bellmore, Kelly Clark, Delia Mauceli, Sammie Travis, Deb Whiting et Beth Young de Dartmouth pour leur assistance administrative. Michael Fromberger, Brian Campbell, Amanda Eubanks, Sung Hoon Kim et Neha Narula nous ont aussi apporté une assistance opportune à Dartmouth.

Nombreux sont celles et ceux qui ont eu la gentillesse de signaler des erreurs dans la première édition. Merci aux personnes suivantes, dont chacune a été la première à signaler une erreur dissimulée dans la première édition : Len Adleman, Selim Akl, Richard Anderson, Juan Andrade-Cetto, Gregory Bachelis, David Barrington, Paul Beame, Richard Beigel, Margrit Betke, Alex Blakemore, Bobby Blumofe, Alexander Brown, Xavier Cazin, Jack Chan, Richard Chang, Chienhua Chen, Ien Cheng, Hoon Choi, Drue Coles, Christian Collberg, George Collins, Eric Conrad, Peter Csaszar, Paul Dietz, Martin Dietzfelbinger, Scot Drysdale, Patricia Ealy, Yaakov Eisenberg, Michael Ernst, Michael Formann, Nedim Fresko, Hal Gabow, Marek Galecki, Igal Galperin, Luisa Gargano, John Gately, Rosario Genario, Mihaly Gereb, Ronald Greenberg, Jerry Grossman, Stephen Guattery, Alexander Hartemik, Anthony Hill, Thomas Hofmeister, Mathew Hostetter, Yih-Chun Hu, Dick Johnsonbaugh, Marcin Jurdzinski, Nabil Kahale, Fumiaki Kamiya, Anand Kanagala, Mark Kantrowitz, Scott Karlin, Dean Kelley, Sanjay Khanna, Haluk Konuk, Dina Kravets, Jon Kroger, Bradley Kuszmaul, Tim Lambert, Hang Lau, Thomas Lengauer, George Madrid, Bruce Maggs, Victor Miller, Joseph Muskat, Tung Nguyen, Michael Orlov, James Park, Seongbin Park, Ioannis Paschalidis, Boaz Patt-Shamir, Leonid Peshkin, Patricio Poblete, Ira Pohl, Stephen Ponzio, Kjell Post, Todd Poynor, Colin Prepscious, Sholom Rosen, Dale Russell, Hershel Safer, Karen Seidel, Joel Seiferas, Erik Seligman, Stanley Selkow, Jeffrey Shallit, Greg Shannon, Micha Sharir, Sasha Shen, Norman Shulman, Andrew Singer, Daniel Sleator, Bob Sloan, Michael Sofka, Volker Strumpfen, Lon Sunshine, Julie Sussman, Asterio Tanaka, Clark Thomborson, Nils Thommesen, Homer Tilton, Martin Tompa, Andrei Toom, Felzer Torsten, Hirendu Vaishnav, M. Veldhorst, Luca Venuti, Jian Wang, Michael Wellman, Gerry Wiener, Ronald Williams, David Wolfe, Jeff Wong, Richard Woundy, Neal Young, Huaiyuan Yu, Tian Yuxing, Joe Zachary, Steve Zhang, Florian Zschoke et Uri Zwick.

Nombre de nos collègues ont fourni des compte-rendus détaillés ou ont rempli un long questionnaire. Merci donc à Nancy Amato, Jim Aspnes, Kevin Compton, William Evans, Peter Gacs, Michael Goldwasser, Andrzej Proskurowski, Vijaya Ramachandran et John Reif. Nous remercions également les personnes suivantes qui nous nous renvoyé le questionnaire : James Abello, Josh Benaloh, Bryan Beresford-Smith, Kenneth Blaha, Hans Bodlaender, Richard Borie, Ted Brown, Domenico Cantone, M. Chen, Robert Cimikowski, William Clocksin, Paul Cull, Rick Decker, Matthew Dickerson, Robert Douglas, Margaret Fleck, Michael Goodrich, Susanne Hambrusch, Dean Hendrix, Richard Johnsonbaugh, Kyriakos Kalorkoti, Srinivas Kankanahalli, Hikyoo Koh, Steven Lindell, Errol Lloyd, Andy Lopez, Dian Rae Lopez, George Lucker, David Maier, Charles Martel, Xiannong Meng, David Mount, Alberto Policriti, Andrzej Proskurowski, Kirk Pruhs, Yves Robert, Guna Seetharaman, Stanley Selkow, Robert Sloan, Charles Steele, Gerard Tel, Murali Varanasi, Bernd Walter et Alden Wright. Nous aurions aimé pouvoir répondre à toutes vos suggestions, mais cette seconde édition aurait dû alors faire dans les 3000 pages !

La seconde édition a été faite avec \LaTeX 2 ϵ . Michael Downes a converti les macros \LaTeX pour les faire passer de \LaTeX « classique » à \LaTeX 2 ϵ ; il a aussi converti les fichiers texte pour qu'ils utilisent les nouvelles macros. David Jones a fourni une assistance sur \LaTeX 2 ϵ . Les figures de la seconde édition ont été réalisées par les auteurs à l'aide de MacDraw Pro. Comme pour la première édition, l'index a été compilé avec Windex, qui est un programme C développé par les auteurs ; la bibliographie a été préparée avec \BIBTeX . Ayorkor Mills-Tettey et Rob Leathern ont aidé à la conversion des figures vers MacDraw Pro ; Ayorkor a, en outre, contrôlé notre bibliographie.

Comme cela avait été le cas pour la première édition, travailler avec The MIT Press et McGraw-Hill fut un vrai plaisir. Nos superviseurs, Bob Prior pour The MIT Press et Betsy Jones pour McGraw-Hill, ont supporté nos caprices et nous ont maintenu dans le droit chemin en maniant adroitement la carotte et le bâton.

Enfin, nous remercions nos femmes (Nicole Cormen, Gail Rivest et Rebecca Ivry), nos enfants (Ricky, William et Debby Leiserson ; Alex et Christopher Rivest ; Molly, Noah et Benjamin Stein) et nos parents (Renee et Perry Cormen ; Jean et Mark Leiserson ; Shirley et Lloyd Rivest ; Irene et Ira Stein) pour leur amour et leur soutien pendant l'écriture de ce livre. La patience et les encouragements de nos familles ont permis à ce projet de voir le jour. Nous leur dédions affectueusement ce livre.

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

*Hanover, New Hampshire
Cambridge, Massachusetts
Cambridge, Massachusetts
Hanover, New Hampshire*

Mai 2001

PARTIE 1

INTRODUCTION

Cette partie présente les fondamentaux qui doivent vous guider pour concevoir et analyser des algorithmes. Elle a pour objectif d'exposer en douceur la manière de spécifier les algorithmes, certaines stratégies de conception qui nous serviront tout au long de ce livre, ainsi que bon nombre des concepts essentiels sous-jacents à l'analyse des algorithmes. Les parties suivantes de ce livre s'appuieront sur toutes ces fondations.

Le chapitre 1 donne une présentation générale des algorithmes et de leur place dans les systèmes informatiques modernes. Ce chapitre définit ce qu'est un algorithme et donne des exemples. Il signale aussi, au passage, que les algorithmes sont une technologie, au même titre que les matériels, les interfaces utilisateur graphique, les systèmes orientés objet ou les réseaux.

Au chapitre 2, nous verrons nos premiers algorithmes, lesquels résolvent le problème qui consiste à trier une suite de n nombres. Ils seront écrits en un pseudo code qui, même s'il n'est pas directement traduisible en quelque langage de programmation traditionnel que ce soit, reflète la structure de l'algorithme de manière suffisamment claire pour qu'un programmeur compétent puisse le mettre en œuvre dans le langage de son choix. Les algorithmes de tri que nous étudierons sont le tri par insertion, qui utilise une stratégie incrémentale, et le tri par fusion, qui utilise une technique récursive baptisée « diviser pour régner ». Bien que la durée d'exécution de chacun de ces algorithmes croisse avec la valeur de n , le taux de croissance n'est pas le même pour les deux algorithmes. Nous déterminerons ces temps d'exécution au chapitre 2 et introduirons une notation très pratique pour les exprimer.

Le chapitre 3 définira plus formellement cette notation, que nous appellerons notation asymptotique. Il commencera par définir plusieurs notations asymptotiques qui nous serviront à borner, à majorer et/ou à minorer, les durées d'exécution des algorithmes. Le reste du chapitre 3 consistera essentiellement en une présentation de notations mathématiques. Le but recherché est de garantir que les notations que vous employez concordent avec celles utilisées dans cet ouvrage, et non de vous enseigner de nouveaux concepts mathématiques.

Le chapitre 4 approfondira la méthode diviser-pour-régner, introduite au chapitre 2. En particulier, le chapitre 4 donnera des méthodes pour la résolution des récurrences, qui sont très utiles pour décrire les durées d'exécution des algorithmes récursifs. Une technique très puissante ici est celle appelée « méthode générale », qui permet de résoudre les récurrences induites par les algorithmes de type diviser-pour-régner. Une bonne partie du chapitre 4 sera consacrée à la démonstration de la justesse de la méthode générale ; vous pouvez, sans inconvénient aucun, sauter cette démonstration.

Le chapitre 5 introduira l'analyse probabiliste et les algorithmes randomisés. L'analyse probabiliste sert généralement à déterminer la durée d'exécution d'un algorithme dans le cas où, en raison de la présence d'une distribution probabiliste intrinsèque, le temps d'exécution peut varier pour différentes entrées de la même taille. Dans certains cas, nous supposerons que les entrées obéissent à une distribution probabiliste connue, de sorte que nous ferons la moyenne des temps d'exécution sur l'ensemble des entrées possibles. Dans d'autres cas, la distribution probabiliste ne viendra pas des entrées, mais de choix aléatoires faits pendant l'exécution de l'algorithme. Un algorithme dont le comportement dépend non seulement de l'entrée, mais aussi de valeurs produites par un générateur de nombres aléatoires est un algorithme randomisé. Nous pouvons employer des algorithmes randomisés pour garantir une distribution probabiliste sur les entrées, et ainsi garantir qu'aucune entrée particulière n'entraîne systématiquement de piètres performances, voire même pour borner les taux d'erreur des algorithmes qui sont autorisés à donner, dans une certaine limite, des résultats erronés.

Les annexes A–C renferment d'autres sujets mathématiques qui vous faciliteront la lecture de cet ouvrage. Vous avez certainement déjà vu une bonne partie de ces notions (bien que les conventions de notation spécifiques que nous utilisons ici puissent, à l'occasion, différer de ce que vous connaissiez) ; vous pouvez donc considérer les annexes comme une sorte de référence. En revanche, la plupart des concepts présentés dans la partie 1 sont vraisemblablement inédits pour vous. Tous les chapitres de la partie 1 et toutes les annexes ont été rédigés dans un esprit très didactique.

Chapitre 1

Rôle des algorithmes en informatique

Qu'est-ce qu'un algorithme ? En quoi l'étude des algorithmes est-elle utile ? Quel est le rôle des algorithmes par rapport aux autres technologies informatiques ? Ce chapitre a pour objectif de répondre à ces questions.

1.1 ALGORITHMES

Voici une définition informelle du terme **algorithme** : procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie.

L'on peut aussi considérer un algorithme comme un outil permettant de résoudre un **problème de calcul** bien spécifié. L'énoncé du problème spécifie, en termes généraux, la relation désirée entre l'entrée et la sortie. L'algorithme décrit une procédure de calcul spécifique permettant d'obtenir cette relation entrée/sortie.

Supposons, par exemple, qu'il faille trier une suite de nombres dans l'ordre croissant. Ce problème, qui revient fréquemment dans la pratique, offre une base fertile pour l'introduction de nombre de techniques de conception et d'outils d'analyse standard. Voici comment nous définissons formellement le **problème de tri** :

Entrée : suite de n nombres $\langle a_1, a_2, \dots, a_n \rangle$.

Sortie : permutation (réorganisation) $\langle a'_1, a'_2, \dots, a'_n \rangle$ de la suite donnée en entrée, de façon que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Ainsi, à partir de la suite $\langle 31, 41, 59, 26, 41, 58 \rangle$, un algorithme de tri produit en sortie la suite $\langle 26, 31, 41, 41, 58, 59 \rangle$. À propos de la suite donnée en entrée, on parle d'**instance** du problème de tri. En général, une **instance d'un problème** consiste en l'entrée (satisfaisant aux contraintes, quelles qu'elles soient, imposées dans l'énoncé du problème) requise par le calcul d'une solution au problème.

Le tri est une opération majeure en informatique (maints programmes l'emploient comme phase intermédiaire), ce qui explique que l'on ait inventé un grand nombre d'algorithmes de tri. L'algorithme optimal pour une application donnée dépend, entre autres facteurs, du nombre d'éléments à trier, de la façon dont les éléments sont plus ou moins triés initialement, des restrictions potentielles concernant les valeurs des éléments, ainsi que du type de périphérique de stockage à utiliser : mémoire principale, disques ou bandes.

Un algorithme est dit **correct** si, pour chaque instance en entrée, il se termine en produisant la bonne sortie. L'on dit qu'un algorithme correct **résout** le problème donné. Un algorithme incorrect risque de ne pas se terminer pour certaines instances en entrée, voire de se terminer sur une réponse autre que celle désirée. Contrairement à ce que l'on pourrait croire, un algorithme incorrect peut s'avérer utile dans certains cas, si son taux d'erreur est susceptible d'être contrôlé. Nous en verrons un exemple au chapitre 31, quand nous étudierons des algorithmes servant à déterminer les grands nombres premiers. En général, seuls nous intéresseront toutefois les algorithmes corrects.

Un algorithme peut être spécifié en langage humain ou en langage informatique, mais peut aussi être basé sur un système matériel. L'unique obligation est que la spécification fournisse une description précise de la procédure de calcul à suivre.

a) Quels sont les types de problème susceptibles d'être résolus par des algorithmes ?

Le tri n'est absolument pas l'unique problème pour lequel ont été inventés des algorithmes. (Vous l'avez sans doute deviné rien qu'en voyant la taille de cet ouvrage.) Les applications concrètes des algorithmes sont innombrables, entre autres :

- Le projet du génome humain a pour objectifs d'identifier les 100 000 gènes de l'ADN humain, de déterminer les séquences des 3 milliards de paires de bases chimiques qui constituent l'ADN humain, de stocker ces informations dans des bases de données et de développer des outils d'analyse de données. Chacune de ces étapes exige des algorithmes très élaborés. Les solutions aux divers problèmes sous-jacents sortent du cadre de ce livre, mais les concepts traités dans nombre de chapitres de cet ouvrage sont utilisés pour résoudre ces problèmes de biologie, permettant ainsi aux scientifiques de faire leur travail tout en utilisant les ressources avec efficacité. Cela fait gagner du temps, aussi bien au niveau des hommes que des machines, et de l'argent, du fait que les expériences de laboratoire peuvent ainsi fournir davantage d'informations.

- Internet permet à des gens éparpillés un peu partout dans le monde d'accéder rapidement à toutes sortes de données. Tout cela repose sur des algorithmes intelligents qui permettent de gérer et manipuler de grosses masses de données. Exemples de problèmes à résoudre : recherche de routes optimales pour l'acheminement des données (ce genre de technique sera présenté au chapitre 24) ; utilisation d'un moteur de recherche pour trouver rapidement les pages contenant tel ou tel type de données (les techniques afférentes seront vues aux chapitres 11 et 32).
- Le commerce électronique permet de négocier et échanger, de manière électronique, biens et services. Le commerce électronique exige que l'on préserve la confidentialité de données telles que numéros de carte de crédit, mots de passe et relevés bancaires. La cryptographie à clé publique et les signatures numériques (traitées au chapitre 31), qui font partie des technologies fondamentales employées dans ce contexte, s'appuient sur des algorithmes numériques et sur la théorie des nombres.
- Dans l'industrie et le commerce, il faut souvent optimiser l'allocation de ressources limitées. Une compagnie pétrolière veut savoir où placer ses puits de façon à maximiser les profits escomptés. Un candidat à la présidence veut savoir dans quels supports publicitaires il doit investir pour maximiser ses chances d'élection. Une compagnie aérienne désire réaliser l'affectation des équipages aux vols de telle façon que les coûts soient minimisés, les vols assurés sans défaillance et la législation respectée. Un fournisseur de services Internet veut savoir où placer des ressources supplémentaires pour desservir ses clients de manière plus efficace. Voilà des exemples de problèmes susceptibles d'être résolus par la programmation linéaire, que nous étudierons au chapitre 29.

Certains détails de ces exemples sortent du cadre de cet ouvrage, mais nous donnerons des techniques qui s'appliquent à ces problèmes et catégories de problème. Nous montrerons aussi, dans ce livre, comment résoudre maints problèmes concrets, dont les suivants :

- Soit une carte routière sur laquelle sont indiquées toutes les distances entre intersections adjacentes ; il faut déterminer le trajet le plus court entre deux intersections. Le nombre d'itinéraires peut être énorme, même si l'on n'a pas d'itinéraire qui se recoupe. Comment trouver le trajet le plus court parmi tous les trajets possibles ? Ici, nous modélisons la carte (qui, elle-même, modélise les routes réelles) sous la forme d'un graphe (sujet traité au chapitre 10 et à l'annexe B), puis nous cherchons à déterminer le chemin le plus court entre deux sommets du graphe. Le chapitre 24 montrera comment résoudre ce problème de manière efficace.
- Soit une suite $\langle A_1, A_2, \dots, A_n \rangle$ de n matrices, dont nous voulons calculer le produit $A_1 A_2 \dots A_n$. La multiplication matricielle étant associative, il existe plusieurs ordres licites pour calculer le produit. Par exemple, si $n = 4$, nous pourrions faire les multiplications comme si le produit était parenthésé de l'une quelconque des façons suivantes : $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$ ou $((A_1A_2)A_3)A_4$. Si toutes les matrices sont carrées (et ont donc la même taille),

l'ordre des multiplications n'affecte pas la durée de calcul du produit. Si, en revanche, les matrices sont de tailles différentes (ces tailles étant, toutefois, compatibles au niveau de la multiplication matricielle), alors l'ordre des multiplications peut faire une très grosse différence. Le nombre d'ordres potentiels de multiplication étant exponentiel en n , essayer tous les ordres possibles risque de demander beaucoup de temps. Nous verrons au chapitre 15 comment utiliser la technique générale connue sous le nom de programmation dynamique pour résoudre ce problème de façon beaucoup plus efficace.

- Soit l'équation $ax = b \pmod{n}$ dans laquelle a , b et n sont des entiers ; nous voulons trouver tous les entiers x , modulo n , qui satisfont à cette équation. Il peut y avoir zéro, une ou plusieurs solutions. Nous pourrions essayer de faire $x = 0, 1, \dots, n-1$ dans l'ordre, mais le chapitre 31 donnera une méthode plus efficace.
- Soient n points du plan, dont nous voulons déterminer l'enveloppe convexe. Cette enveloppe est le plus petit polygone convexe qui contient les points. Intuitivement, nous pouvons nous représenter chaque point comme un clou planté dans une planche. L'enveloppe convexe serait alors représentée par un élastique qui entoure tous les clous. Chaque clou autour duquel s'enroule l'élastique est un sommet de l'enveloppe convexe. (Voir la figure 33.6 pour un exemple.) N'importe lequel des 2^n sous-ensembles des points pourrait correspondre aux sommets de l'enveloppe convexe. Seulement, il ne suffit pas de savoir quels sont les points qui sont des sommets de l'enveloppe ; il faut aussi connaître l'ordre dans lequel ils apparaissent. Il existe donc bien des choix pour les sommets de l'enveloppe convexe. Le chapitre 33 donnera deux bonnes méthodes pour la détermination de l'enveloppe convexe.

Ces énumérations sont loin d'être exhaustives (comme vous l'avez probablement deviné en soupesant ce livre), mais elles témoignent de deux caractéristiques que l'on retrouve dans bon nombre d'algorithmes intéressants :

- 1) Il existe beaucoup de solutions à priori, mais la plupart d'entre elles ne sont pas celles que nous voulons. Trouver une solution qui convienne vraiment, voilà qui n'est pas toujours évident.
- 2) Il y a des applications concrètes. Parmi les problèmes précédemment énumérés, les chemins les plus courts fournissent les exemples les plus élémentaires. Une entreprise de transport, par exemple une société de camionnage ou la SNCF, a intérêt à trouver les trajets routiers ou ferroviaires les plus courts, car cela diminue les coûts de main d'œuvre et d'énergie. Un nœud de routage Internet peut avoir à déterminer le chemin le plus court à travers le réseau pour minimiser le délai d'acheminement d'un message.

b) Structures de données

Cet ouvrage présente plusieurs **structures de données**. Une structure de données est un moyen de stocker et organiser des données pour faciliter l'accès à ces données et leur modification. Il n'y a aucune structure de données qui réponde à tous les

besoins, de sorte qu'il importe de connaître les forces et limitations de plusieurs de ces structures.

c) *Technique*

Vous pouvez vous servir de cet ouvrage comme d'un « livre de recettes » pour algorithmes, mais vous risquez tôt ou tard de tomber sur un problème pour lequel il n'existe pas d'algorithme publié (c'est le cas, par exemple, de nombre des exercices et problèmes donnés dans ce livre !). Cet ouvrage vous enseignera des techniques de conception et d'analyse d'algorithme, de façon que vous puissiez créer des algorithmes de votre cru, prouver qu'ils fournissent la bonne réponse et comprendre leur efficacité.

d) *Problèmes difficiles*

Une bonne partie de ce livre concerne les algorithmes efficaces. Notre mesure habituelle de l'efficacité est la vitesse, c'est-à-dire la durée que met un algorithme à produire ses résultats. Il existe des problèmes, cependant, pour lesquels l'on ne connaît aucune solution efficace. Le chapitre 34 étudie un sous-ensemble intéressant de ces problèmes, connus sous l'appellation de problèmes NP-complets.

En quoi les problèmes NP-complets sont-ils intéressants ? Primo, l'on n'a jamais trouvé d'algorithme efficace pour un problème NP-complet, mais personne n'a jamais prouvé qu'il ne peut pas exister d'algorithme efficace pour un problème. Autrement dit, l'on ne sait pas s'il existe ou non des algorithmes efficaces pour les problèmes NP-complets. Secundo, l'ensemble des problèmes NP-complets offre la propriété remarquable suivante : s'il existe un algorithme efficace pour l'un quelconque de ces problèmes, alors il existe des algorithmes efficaces pour tous. Cette relation entre les problèmes NP-complets rend d'autant plus frustrante l'absence de solutions efficaces. Tertio, plusieurs problèmes NP-complets ressemblent, sans être identiques, à des problèmes pour lesquels nous connaissons des algorithmes efficaces. Un petit changement dans l'énoncé du problème peut entraîner un changement majeur au niveau de l'efficacité du meilleur algorithme connu.

Il est intéressant d'avoir quelques connaissances concernant les problèmes NP-complets étant donné que, chose surprenante, certains d'entre eux reviennent souvent dans les applications concrètes. Si l'on vous demande de concocter un algorithme efficace pour un problème NP-complet, vous risquez de perdre pas mal de temps à chercher pour rien. Si vous arrivez à montrer que le problème est NP-complet, vous pourrez alors consacrer votre temps à développer un algorithme efficace fournissant une solution qui est bonne sans être optimale.

À titre d'exemple concret, prenons le cas d'une entreprise de camionnage ayant un dépôt central. Chaque jour, elle charge le camion au dépôt puis l'envoie faire des livraisons à plusieurs endroits. À la fin de la journée, le camion doit revenir au dépôt de façon à pouvoir être rechargé le jour suivant. Pour réduire les coûts, l'entreprise

veut choisir un ordre de livraisons tel que la distance parcourue par le camion soit minimale. Ce problème, qui n'est autre que le fameux « problème du voyageur de commerce », est un problème NP-complet. Il n'a pas d'algorithme efficace connu. Sous certaines hypothèses, cependant, il existe des algorithmes efficaces donnant une distance globale qui n'est pas trop éloignée de la distance minimale. Le chapitre 35 présente ce genre d'algorithmes, dits « algorithmes d'approximation ».

Exercices

1.1.1 * Donnez un exemple concret qui intègre l'un des problèmes suivants : tri, optimisation de l'ordre de multiplication des matrices, détermination de l'enveloppe convexe.

1.1.2 * À part la vitesse, qu'est-ce qui pourrait servir à mesurer l'efficacité dans un contexte concret ?

1.1.3 * Sélectionnez une structure de données que vous avez déjà vue, puis étudiez ses avantages et ses inconvénients.

1.1.4 * En quoi le problème du chemin minimal et celui du voyageur de commerce, précédemment mentionnés, se ressemblent-ils ? En quoi sont-ils différents ?

1.1.5 * Trouvez un problème concret pour lequel seule conviendra la solution optimale. Trouvez ensuite un problème pour lequel une solution « approchée » pourra faire l'affaire.

1.2 ALGORITHMES EN TANT QUE TECHNOLOGIE

Supposez que les ordinateurs soient infiniment rapides et que leurs mémoires soient gratuites. Faudrait-il encore étudier les algorithmes ? Oui, ne serait-ce que pour montrer que la solution ne boucle pas indéfiniment et qu'elle se termine avec la bonne réponse.

Si les ordinateurs étaient infiniment rapides, n'importe quelle méthode correcte de résolution d'un problème ferait l'affaire. Vous voudriez sans doute que votre solution entre dans le cadre d'une bonne méthodologie d'ingénierie (c'est-à-dire qu'elle soit bien conçue et bien documentée), mais vous privilégieriez le plus souvent la méthode qui est la plus simple à mettre en œuvre.

Si rapides que puissent être les ordinateurs, ils ne sont pas infiniment rapides. Si bon marché que puisse être la mémoire, elle n'est pas gratuite. Le temps machine est donc une ressource limitée, et il en est de même de l'espace mémoire. Il faut utiliser ces ressources avec parcimonie, et des algorithmes performants, en termes de durée et d'encombrement, vous aideront à atteindre cet objectif.

a) *Efficacité*

Il arrive souvent que des algorithmes conçus pour résoudre le même problème diffèrent fortement entre eux en termes d'efficacité. Ces différences peuvent être bien plus importantes que celles dues au matériel et au logiciel.

À titre d'exemple, au chapitre 2 nous verrons deux algorithmes de tri. Le premier, appelé **tri par insertion**, prend un temps approximativement égal à $c_1 n^2$ pour trier n éléments, c_1 étant une constante indépendante de n . La durée du tri est donc, grosso modo, proportionnelle à n^2 . Le second, appelé **tri par fusion**, prend un temps approximativement égal à $c_2 n \lg n$, où $\lg n$ désigne $\log_2 n$ et c_2 est une autre constante indépendante, elle aussi, de n . Le tri par insertion a généralement un facteur constant inférieur à celui du tri par fusion, de sorte que $c_1 < c_2$. Nous verrons que les facteurs constants peuvent être beaucoup moins significatifs, au niveau de la durée d'exécution, que la dépendance par rapport au nombre n de données à trier. Là où le tri par fusion a un facteur $\lg n$ dans son temps d'exécution, le tri par insertion a un facteur n , qui est beaucoup plus grand. Le tri par insertion est généralement plus rapide que le tri par fusion pour de petits nombres de données mais, dès que le nombre n d'éléments à trier devient suffisamment grand, l'avantage du tri par fusion ($\lg n$ contre n) fait plus que compenser la différence entre les facteurs constants. Même si c_1 est très inférieur à c_2 , il y a toujours un palier au-delà duquel le tri par fusion devient plus rapide.

Pour avoir un exemple concret, comparons un ordinateur rapide (ordinateur A) exécutant un tri par insertion et un ordinateur lent (ordinateur B) exécutant un tri par fusion. Ces deux machines doivent chacune trier un tableau d'un million de nombres. Supposez que l'ordinateur A exécute un milliard d'instructions par seconde et que l'ordinateur B n'exécute que dix millions d'instructions par seconde, de sorte que l'ordinateur A est 100 fois plus rapide que l'ordinateur B en termes de puissance de calcul brute. Pour rendre la différence encore plus sensible, supposez que le meilleur programmeur du monde écrive le tri par insertion en langage machine pour l'ordinateur A et que le code résultant demande $2n^2$ instructions pour trier n nombres. (Ici, $c_1 = 2$.) Le tri par fusion, en revanche, est programmé pour l'ordinateur B par un programmeur médiocre utilisant un langage de haut niveau avec un compilateur peu performant, de sorte que le code résultant demande $50n \lg n$ instructions (donc, $c_2 = 50$). Pour trier un million de nombres, l'ordinateur A demande

$$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/seconde}} = 2\,000 \text{ secondes,}$$

alors que l'ordinateur B demande

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/seconde}} \approx 100 \text{ secondes,}$$

Avec un algorithme dont le temps d'exécution croît plus lentement, même si le compilateur est médiocre la machine B tourne 20 fois plus vite que la machine

A ! L'avantage du tri par fusion ressort encore plus si nous trions dix millions de nombres : là où le tri par insertion demande environ 2,3 jours, le tri par fusion prend moins de 20 minutes. En général, plus la taille du problème augmente et plus augmente aussi l'avantage relatif du tri par fusion.

b) Algorithmes et autres technologies

L'exemple précédent montre que les algorithmes, à l'instar des matériels informatiques, sont une **technologie**. Les performances globales du système dépendent autant des algorithmes que des matériels. Comme toutes les autres technologies informatiques, les algorithmes ne cessent de progresser.

L'on peut se demander si les algorithmes sont vraiment si importants que cela sur les ordinateurs modernes, compte tenu de l'état d'avancement d'autres technologies telles que

- horloges ultra-rapides, pipelines et architectures superscalaires,
- interfaces utilisateur graphiques conviviales et intuitives,
- systèmes orientés objet, et
- technologies de réseau local et de réseau étendu.

La réponse est oui. Même si certaines applications n'exigent pas explicitement d'algorithmes au niveau de l'application elle-même (c'est le cas, par exemple, de certaines applications simples basées sur le web), la plupart intègrent une certaine dose intrinsèque d'algorithmes. Prenez le cas d'un service basé sur le web qui détermine des trajets pour aller d'un endroit à un autre. (Ce genre de service existe déjà à l'heure où nous écrivons ces lignes.) Sa mise en œuvre exige des matériels rapides, une interface utilisateur graphique, des technologies de réseau étendu, voire des fonctionnalités objet. À cela s'ajoutent des algorithmes pour certains traitements tels que recherche d'itinéraires (utilisant vraisemblablement un algorithme de chemin minimal), dessin de cartes et interpolation d'adresses.

Qui plus est, même une application qui n'emploie pas d'algorithmes au niveau de l'application elle-même s'appuie indirectement sur une foule d'algorithmes. L'application s'exécute sur des matériels performants ? La conception de ces matériels a utilisé des algorithmes. L'application tourne par dessus des interfaces graphiques ? Toutes les interfaces utilisateur graphiques reposent sur des algorithmes. L'application fonctionne en réseau ? Le routage s'appuie fondamentalement sur des algorithmes. L'application a été écrite dans un langage autre que du code machine ? Alors, elle a été traduite par un compilateur, un interpréteur ou un assembleur, toutes ces belles choses faisant un usage intensif d'algorithmes. Les algorithmes sont au cœur de la plupart des technologies employées dans les ordinateurs modernes.

Enfin, avec les capacités sans cesse accrues des ordinateurs, ces derniers traitent des problèmes de plus en plus vastes. Comme nous l'avons vu dans la comparaison

entre le tri par insertion et le tri par fusion, c'est avec les problèmes de grande taille que les différences d'efficacité entre algorithmes sont les plus flagrantes.

Posséder une base solide en algorithmique, voilà qui fait toute la différence entre le programmeur d'élite et le programmeur lambda. Les technologies informatiques modernes permettent de faire certaines tâches même si l'on ne s'y connaît guère en algorithmes ; mais avec une bonne formation en algorithmique, l'on peut aller beaucoup, beaucoup plus loin.

Exercices

1.2.1 Donnez un exemple d'application exigeant des algorithmes intrinsèques, puis discutez les fonctions des algorithmes concernés.

1.2.2 On veut comparer les implémentations du tri par insertion et du tri par fusion sur la même machine. Pour un nombre n d'éléments à trier, le tri par insertion demande $8n^2$ étapes alors que le tri par fusion en demande $64n \lg n$. Quelles sont les valeurs de n pour lesquelles le tri par insertion l'emporte sur le tri par fusion ?

1.2.3 Quelle est la valeur minimale de n pour laquelle un algorithme dont le temps d'exécution est $100n^2$ s'exécute plus vite qu'un algorithme dont le temps d'exécution est $2n$ sur la même machine ?

PROBLÈMES

1.1. Comparaison de temps d'exécution

Pour chaque fonction $f(n)$ et pour chaque durée t du tableau suivant, déterminez la taille maximale n d'un problème susceptible d'être résolu dans le temps t , en supposant que l'algorithme mette $f(n)$ microsecondes pour traiter le problème.

	1 seconde	1 minute	1 heure	1 jour	1 mois	1 an	1 siècle
$\lg n$							
\sqrt{n}							
n							
$n \lg n$							
n^2							
n^3							
$2n$							
$n!$							

NOTES

Il existe nombre de textes excellents sur le thème général des algorithmes, dont les ouvrages de Aho, Hopcroft et Ullman [5, 6], Baase et Van Gelder [26], Brassard et Bratley [46, 47], Goodrich et Tamassia [128], Horowitz, Sahni et Rajasekaran [158], Kingston [179], Knuth [182, 183, 185], Kozen [193], Manber [210], Mehlhorn [217, 218, 219], Purdom et Brown [252], Reingold, Nievergelt et Deo [257], Sedgewick [269], Skiena [280] et Wilf [315]. Certains des aspects plus concrets de la conception des algorithmes sont traités par Bentley [39, 40] et Gonnet [126]. Vous trouverez aussi une introduction à l'algorithmique dans les ouvrages intitulés *Handbook of Theoretical Computer Science, Volume A* [302] et *CRC Handbook on Algorithms and Theory of Computation* [24]. Enfin, vous trouverez des présentations d'algorithmes utilisés en biologie informatique dans les manuels de Gusfield [136], Pevzner [240], Setubal et Medinas [272] et Waterman [309].

Chapitre 2

Premiers pas

Ce chapitre a pour but de vous familiariser avec la démarche que nous adopterons dans ce livre quand il s'agira de réfléchir à la conception et à l'analyse des algorithmes. On peut le lire indépendamment de la suite, bien qu'il fasse plusieurs fois référence à des notions qui seront abordées dans les chapitres 3 et 4. (Il contient aussi plusieurs sommations que l'annexe A montrera comment résoudre).

Nous commencerons par étudier l'algorithme du tri par insertion, afin de résoudre la problématique du tri exposée au chapitre 1. Nous introduirons un « pseudo code » qui ne devrait pas surprendre le lecteur ayant déjà pratiqué la programmation, et qui nous servira à spécifier nos algorithmes. Après avoir décrit en pseudo code l'algorithme du tri par insertion, nous montrerons qu'il fait bien ce qu'on attend de lui et nous analyserons son temps d'exécution. L'analyse permettra d'introduire une notation qui met en valeur la façon dont le temps d'exécution croît avec le nombre d'éléments à trier. Après avoir étudié le tri par insertion, nous présenterons le paradigme « diviser pour régner » pour la conception d'algorithmes, technique qui nous servira à développer l'algorithme du tri par fusion. Nous terminerons par l'analyse du temps d'exécution du tri par fusion.

2.1 TRI PAR INSERTION

Notre premier algorithme, à savoir le tri par insertion, résout la problématique du tri exposée au chapitre 1 :

Entrée : Suite de n nombres $\langle a_1, a_2, \dots, a_n \rangle$.

Sortie : permutation (réorganisation) $\langle a'_1, a'_2, \dots, a'_n \rangle$ de la suite donnée en entrée, de façon que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Les nombres à trier sont parfois appelés *clés*.

Dans cet ouvrage, nous exprimerons généralement les algorithmes sous la forme de programmes écrits en un *pseudo code* qui, à certains égards, rappelle C, Pascal ou Java. Si vous connaissez déjà l'un de ces langages, vous n'aurez guère de mal à lire nos algorithmes. Ce qui différencie le pseudo code du « vrai » code c'est que, avec le pseudo code nous employons l'écriture qui nous semble être la plus claire et la plus concise pour spécifier l'algorithme ; ne soyez donc pas surpris de voir apparaître un mélange de français et de « vrai » code. Autre différence entre pseudo code et vrai code : le pseudo code ne se soucie pas, en principe, de problèmes d'ingénierie logicielle tels qu'abstraction des données, modularité, traitement d'erreur, etc. Cela permet au pseudo code de refléter plus clairement la substance moelle de l'algorithme.

Nous commencerons par le *tri par insertion*, qui est un algorithme efficace quand il s'agit de trier un petit nombre d'éléments. Le tri par insertion s'inspire de la manière dont la plupart des gens tiennent des cartes à jouer. Au début, la main gauche du joueur est vide et ses cartes sont posées sur la table. Il prend alors sur la table les cartes, une par une, pour les placer dans sa main gauche. Pour savoir où placer une carte dans son jeu, le joueur la compare avec chacune des cartes déjà présentes dans sa main gauche, en examinant les cartes de la droite vers la gauche, comme le montre la figure 2.1. A tout moment, les cartes tenues par la main gauche sont triées ; ces cartes étaient, à l'origine, les cartes situées au sommet de la pile sur la table.

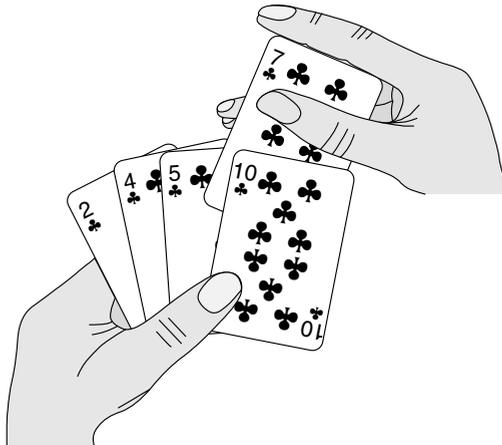


Figure 2.1 Tri de cartes à jouer, via tri par insertion.

Notre pseudo-code pour le tri par insertion se présente sous la forme d'une procédure appelée TRI-INSERTION. Elle prend comme paramètre un tableau $A[1..n]$ qui

contient une séquence à trier, de longueur n . (Dans le code, le nombre n d'éléments de A est noté $longueur[A]$). Les nombres donnés en entrée sont *triés sur place* : ils sont réorganisés à l'intérieur du tableau A , avec tout au plus un nombre constant d'entre eux stocké à l'extérieur du tableau à tout instant. Lorsque TRI-INSERTION se termine, le tableau d'entrée A contient la séquence de sortie triée.

TRI-INSERTION(A)

```

1  pour  $j \leftarrow 2$  à  $longueur[A]$ 
2      faire  $clé \leftarrow A[j]$ 
3          ▷ Insère  $A[j]$  dans la séquence triée  $A[1..j-1]$ .
4           $i \leftarrow j - 1$ 
5          tant que  $i > 0$  et  $A[i] > clé$ 
6              faire  $A[i + 1] \leftarrow A[i]$ 
7               $i \leftarrow i - 1$ 
8           $A[i + 1] \leftarrow clé$ 

```

a) Invariants de boucle et validité du tri par insertion

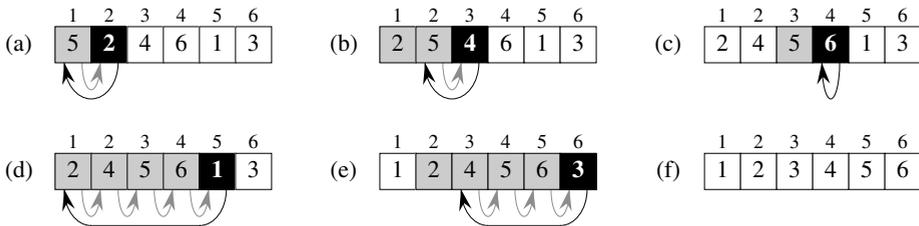


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = (5, 2, 4, 6, 1, 3)$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle **pour** des lignes 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f) Tableau trié final.

La figure 2.2 illustre le fonctionnement de cet algorithme pour $A = (5, 2, 4, 6, 1, 3)$. L'indice j indique la « carte courante » en cours d'insertion dans la main. Au début de chaque itération de la boucle **pour** « extérieure », indexée par j , le sous-tableau composé des éléments $A[1..j-1]$ correspond aux cartes qui sont déjà dans la main, alors que les éléments $A[j+1..n]$ correspondent aux cartes qui sont encore sur la table. En fait, les éléments $A[1..j-1]$ sont les éléments qui occupaient *initialement* les positions 1 à $j-1$, mais qui depuis ont été triés. Nous utiliserons ces propriétés de $A[1..j-1]$ pour définir ce que l'on appelle, de manière plus formelle, un *invariant de boucle* : Au début de chaque itération de la boucle **pour** des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.

Les invariants de boucle nous aideront à comprendre pourquoi un algorithme est correct. Nous devons montrer trois choses, concernant un invariant de boucle :

Initialisation : Il est vrai avant la première itération de la boucle.

Conservation : S'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.

Terminaison : Une fois terminée la boucle, l'invariant fournit une propriété utile qui aide à montrer la validité de l'algorithme.

Si les deux premières propriétés sont vérifiées, alors l'invariant est vrai avant chaque itération de la boucle. Notez la ressemblance avec la récurrence mathématique dans laquelle, pour démontrer qu'une propriété est vraie, vous montrez, primo qu'elle est vraie pour une valeur initiale, secundo que, si elle est vraie pour le rang N , alors elle l'est pour le rang $N + 1$ (phase inductive).

La troisième propriété est peut-être la plus importante, vu que nous utilisons l'invariant de boucle pour prouver la validité de l'algorithme. Elle diffère aussi de l'usage habituel de la récurrence mathématique, dans laquelle la phase inductive se répète indéfiniment ; ici, on arrête « l'induction » quand la boucle se termine.

Voyons comment ces propriétés s'appliquent au tri par insertion.

Initialisation : Commençons par montrer que l'invariant est vérifié avant la première itération de la boucle, quand $j = 2$.⁽¹⁾ Le sous-tableau $A[1 \dots j-1]$ se compose donc uniquement de l'élément $A[1]$ qui est, en fait, l'élément originel de $A[1]$. En outre, ce sous-tableau est trié (c'est une trivialité), ce qui montre bien que l'invariant est vérifié avant la première itération de la boucle.

Conservation : Passons ensuite à la deuxième propriété : montrer que chaque itération conserve l'invariant. De manière informelle, le corps de la boucle **pour** extérieure fonctionne en déplaçant $A[j-1]$, $A[j-2]$, $A[j-3]$, etc. d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour $A[j]$ (lignes 4–7), auquel cas on insère la valeur de $A[j]$ (ligne 8). Un traitement plus formel de la deuxième propriété nous obligerait à formuler et prouver un invariant pour la boucle **tant que** « intérieure ». Pour l'instant, nous ne nous encombrons pas d'un tel formalisme et nous nous appuyerons uniquement sur notre analyse informelle pour montrer que la deuxième propriété est vérifiée pour la boucle extérieure.

Terminaison : Enfin, voyons ce qui se passe quand la boucle se termine. Pour le tri par insertion, la boucle **pour** extérieure prend fin quand j dépasse n , c'est-à-dire quand $j = n+1$. En substituant $n+1$ à j dans la formulation de l'invariant de boucle, l'on a que le sous-tableau $A[1 \dots n]$ se compose des éléments qui appartenaient originellement à $A[1 \dots n]$ mais qui ont été triés depuis lors. Or, le sous-tableau

(1) Quand la boucle est une boucle **pour**, le moment où nous contrôlons l'invariant avant la première itération se situe juste après l'assignation initiale à la variable servant de compteur de boucle et juste avant le premier test dans l'en-tête de boucle. Dans le cas de TRI-INSERTION, cet instant se situe après affectation de la valeur 2 à la variable j mais avant le premier test vérifiant si $j \leq \text{longueur}[A]$.

$A[1..n]$ n'est autre que le tableau complet ! Par conséquent, le tableau tout entier est trié, et donc l'algorithme est correct.

Nous reverrons plus loin dans ce chapitre, ainsi que dans d'autres chapitres, cet emploi des invariants de boucle pour justifier la validité des algorithmes.

b) Conventions concernant le pseudo code

Nous adopterons les conventions suivantes pour le pseudo code.

- 1) L'indentation indique une structure de bloc. Par exemple le corps de la boucle **pour** qui commence à la ligne 1 se compose des lignes 2–8, et le corps de la boucle **tant que** qui commence à la ligne 5 contient les lignes 6–7 mais pas la ligne 8. Notre style d'indentation s'applique également aux instructions **si-alors-sinon**. L'emploi de l'indentation, au lieu d'indicateurs du genre **début** et **fin**, pour matérialiser les blocs réduit sensiblement l'encombrement tout en préservant, voire améliorant, la clarté. ⁽²⁾.
- 2) Les boucles **tant que**, **pour** et **répéter**, ainsi que tests **si**, **alors** et **sinon** ont la même signification qu'en Pascal. ⁽³⁾ Il existe, toutefois, une différence subtile concernant les boucles **pour** : en Pascal la valeur de la variable servant de compteur de boucle est indéfinie à la sortie de la boucle, alors que dans ce livre le compteur de boucle conserve sa valeur après la fin de la boucle. Ainsi donc, juste après une boucle **pour**, la valeur du compteur de boucle est la valeur immédiatement supérieure à la limite de bouclage. Nous avons utilisé cette propriété dans notre démonstration de la conformité du tri par insertion. L'en-tête de la boucle **pour**, en ligne 1, est **pour** $j \leftarrow 2$ à $\text{longueur}[A]$; donc, quand la boucle se termine, $j = \text{longueur}[A] + 1$ (ou, de manière équivalente, $j = n + 1$, puisque $n = \text{longueur}[A]$).
- 3) Le symbole « \triangleright » indique que le reste de la ligne est un commentaire.
- 4) Une affectation multiple de la forme $i \leftarrow j \leftarrow e$ affecte aux deux variables i et j la valeur de l'expression e ; on la considère comme équivalente à l'affectation $j \leftarrow e$ suivie de l'affectation $i \leftarrow j$.
- 5) Les variables comme i , j et clé sont locales à la procédure donnée. Nous n'utiliserons pas de variables globales, sauf indication explicite.
- 6) On accède aux éléments d'un tableau via le nom du tableau suivi de l'indice entre crochets. Ainsi, $A[i]$ représente le i -ème élément du tableau A . La notation « \dots » indique une plage de valeurs dans un tableau. $A[1..j]$ représente donc le sous-tableau de A constitué des éléments $A[1], A[2], \dots, A[j]$.

(2) Dans un vrai langage de programmation, il est généralement déconseillé d'utiliser la seule indentation pour représenter les blocs, car il est difficile de repérer les niveaux d'indentation lorsque le code tient sur plusieurs pages.

(3) La plupart des langages structurés ont des constructions équivalentes, sachant que la syntaxe exacte peut différer de celle de Pascal.

7) Les données composées sont le plus souvent organisées en **objets**, constitués d'**attributs** ou **champs**. On accède à un champ particulier grâce à son nom, suivi du nom de l'objet entre crochets. Par exemple, on considérera un tableau comme un objet qui a un attribut *longueur* indiquant le nombre d'éléments contenus dans l'objet. Pour spécifier le nombre d'éléments d'un tableau A , on écrit $\text{longueur}[A]$. Bien que les crochets s'utilisent tant pour l'indexation des tableaux que pour l'accès aux attributs des objets, le contexte indique généralement comment on doit les interpréter.

Une variable représentant un tableau ou un objet est considérée comme un pointeur vers les données représentant le tableau ou l'objet. Pour tous les champs f d'un objet x , faire $y \leftarrow x$ entraîne que $f[y] = f[x]$. Par ailleurs, si l'on fait maintenant $f[x] \leftarrow 3$, on a ensuite non seulement $f[x] = 3$ mais aussi $f[y] = 3$. Autrement dit, x et y pointent vers le même objet après l'affectation $y \leftarrow x$.

Il arrive qu'un pointeur ne fasse référence à aucun objet. Nous lui donnons dans ce cas la valeur particulière NIL.

8) Les paramètres sont passés à une procédure **par valeur** : la procédure appelée reçoit son propre exemplaire des paramètres et, si elle modifie la valeur d'un paramètre, le changement n'est *pas* visible pour la routine appelante. Lorsque les paramètres sont des objets, il y a copie du pointeur pointant vers les données représentant l'objet mais il n'y a pas copie des champs de l'objet. Par exemple, si x est un paramètre d'une procédure appelée, l'affectation $x \leftarrow y$ à l'intérieur de la procédure appelée n'est pas visible pour la procédure appelante. En revanche, l'affectation $f[x] \leftarrow 3$ est visible.

9) Les opérateurs booléens « et » et « ou » sont **court-circuitants**. Cela signifie que, quand on évalue l'expression « x et y », on commence par évaluer x . Si x vaut FAUX, alors l'expression globale ne peut pas être égale à VRAI et il est donc inutile d'évaluer y . Si, en revanche, x vaut VRAI, alors il faut évaluer y pour déterminer la valeur de l'expression globale. De même, dans l'expression « x ou y » on n'évalue y que si x vaut FAUX. Les opérateurs court-circuitants nous permettent d'écrire des expressions booléennes du genre « $x \neq \text{NIL}$ et $f[x] = y$ » sans que nous ayons à nous soucier de ce qui se passerait si on essayait d'évaluer $f[x]$ quand x vaut NIL.

Exercices

2.1.1 À l'aide de la figure 2.2, illustrer l'action de TRI-INSERTION sur le tableau $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1.2 Réécrire la procédure TRI-INSERTION pour trier dans l'ordre non croissant et non dans l'ordre non décroissant.

2.1.3 Considérez le **problème de la recherche** :

Entrée : Une suite de n nombres $A = \langle a_1, a_2, \dots, a_n \rangle$ et une valeur v .

Sortie : Un indice i tel que $v = A[i]$, ou bien la valeur spéciale NIL si v ne figure pas dans A .

Écrire du pseudo code pour *recherche linéaire*, qui parcourt la suite en cherchant v . En utilisant un invariant de boucle, montrer la validité de l'algorithme. Vérifier que l'invariant possède bien les trois propriétés requises.

2.1.4 On considère le problème consistant à additionner deux entiers en représentation binaire stockés sur n bits, rangés dans deux tableaux A et B à n éléments. La somme des deux entiers doit être stockée sous forme binaire dans un tableau C à $n + 1$ éléments. Énoncer le problème formellement et écrire du pseudo-code pour additionner les deux entiers.

2.2 ANALYSE DES ALGORITHMES

Analyser un algorithme est devenu synonyme de prévoir les ressources nécessaires à cet algorithme. Parfois, les ressources à prévoir sont la mémoire, la largeur de bande d'une communication ou le processeur ; mais, le plus souvent, c'est le temps de calcul qui nous intéresse. En général, en analysant plusieurs algorithmes susceptibles de résoudre le problème, on arrive aisément à identifier le plus efficace. Ce type d'analyse peut révéler plus d'un candidat viable, mais parvient généralement à éliminer les algorithmes inférieurs.

Pour pouvoir analyser un algorithme, il faut avoir un modèle de la technologie qui sera employée, notamment un modèle pour les ressources de cette technologie et pour leurs coûts. Dans la majeure partie de ce livre, on supposera que l'on a un modèle de calcul générique basé sur une *machine à accès aléatoire (RAM)* à processeur unique. On devra également avoir à l'esprit que nos algorithmes seront implémentés en tant que programmes informatiques. Dans le modèle RAM, les instructions sont exécutées l'une après l'autre, sans opérations simultanées. Dans des chapitres ultérieurs, cependant, nous aurons l'occasion d'examiner des modèles pour matériel numérique.

À strictement parler, il faudrait définir précisément les instructions du modèle RAM et leurs coûts. Cependant, cela serait pénible et n'apporterait pas grand chose en matière de conception et d'analyse d'algorithme. Attention, toutefois, à ne pas enfreindre allègrement le modèle RAM. Par exemple, que se passerait-il si une RAM avait une instruction de tri ? Alors, on pourrait trier avec une seule instruction. Une telle RAM serait irréaliste, vu que les ordinateurs ne disposent pas de ce genre d'instructions. Nous devons donc nous laisser guider par le fonctionnement des ordinateurs concrets. Le modèle RAM contient les instructions que l'on trouve dans la plupart des ordinateurs : arithmétique (addition, soustraction, multiplication, division, modulo, partie entière, partie entière supérieure, transfert de données (lecture, stockage, copie) et instructions de contrôle (branchement conditionnel et inconditionnel, appel de sous-routine et sortie de sous-routine). Chacune de ces instructions a un temps d'exécution constant.

Les types de données du modèle RAM sont le type entier et le type virgule flottante. Nous ne nous soucierons généralement pas de la précision dans cet ouvrage

mais, pour certaines applications, la précision est un élément crucial. Nous supposons aussi qu'il existe une limite à la taille de chaque mot de données. Ainsi, quand nous travaillerons avec des entrées de taille n , nous supposons en principe que les entiers sont représentés par $c \lg n$ bits pour une certaine constante $c \geq 1$. Nous imposons que $c \geq 1$ de façon que chaque mot puisse contenir la valeur n , ce qui nous permettra d'indexer les divers éléments de l'entrée ; nous imposons aussi à c d'être une constante de façon que la taille du mot n'augmente pas de manière arbitraire. (Si la taille du mot pouvait croître de manière arbitraire, alors on pourrait stocker un volume énorme de données dans un seul mot et manipuler ce volume de données en un temps constant, ce qui est un scénario manifestement irréaliste.)

Les ordinateurs réels contiennent des instructions qui ne sont pas citées ici, et ces instructions supplémentaires représentent une zone d'ombre du modèle RAM. Par exemple, est-ce que l'exponentiation est une instruction à délai constant ? En général, non ; elle exige plusieurs instructions pour calculer x^y quand x et y sont des réels. Dans certains cas particuliers, cependant, l'exponentiation est une opération à délai constant. Maints ordinateurs disposent d'une instruction de « décalage vers la gauche » qui, en temps constant, décale les bits d'un entier de k positions vers la gauche. Pour la plupart des ordinateurs, décaler les bits d'un entier d'une position vers la gauche revient à faire une multiplication par 2. Décaler les bits de k positions vers la gauche revient donc à multiplier par 2^k . Par conséquent, ce genre de machine peut calculer 2^k en une unique instruction à délai constant, et ce en décalant l'entier 1 de k positions vers la gauche, du moment que k n'est pas supérieur au nombre de bits d'un mot machine. Nous ferons tous nos efforts pour éviter ce genre de zone d'ombre du modèle RAM, mais nous considérerons le calcul de 2^k comme une opération à délai constant quand k est un entier positif suffisamment petit.

Dans le modèle RAM, nous n'essayons pas de refléter la hiérarchisation de la mémoire que l'on trouve généralement dans les ordinateurs modernes. Cela veut dire que nous ne modélisons pas les caches, ni la mémoire virtuelle (qui est très souvent mise en œuvre avec la pagination à la demande). Il existe plusieurs modèles de calcul qui essaient de prendre en compte les effets de la hiérarchisation de la mémoire, effets parfois significatifs dans la pratique. Quelques rares problèmes, dans cet ouvrage, examinent les effets de la hiérarchisation de la mémoire mais, dans l'ensemble, les analyses faites dans ce livre n'en tiennent pas compte. Les modèles intégrant la hiérarchisation de la mémoire sont nettement plus compliqués que le modèle RAM, de sorte que leur utilisation risque d'être délicate. Qui plus est, les analyses basées sur le modèle RAM donnent généralement d'excellentes prévisions quant aux performances obtenues sur les machines réelles.

L'analyse d'un algorithme, même simple, dans le modèle RAM peut s'avérer complexe. On devra parfois faire appel à des outils mathématiques tels que l'algèbre combinatoire et la théorie des probabilités ; en outre, il faudra être capable de jongler avec l'algèbre et de repérer les termes les plus significatifs dans une formule. Sachant que le comportement d'un algorithme peut varier pour chaque entrée possible, il nous

faut un moyen de résumer ce comportement en quelques formules simples et faciles à comprendre.

Bien qu'on choisisse le plus souvent un seul modèle de machine pour analyser un algorithme donné, on est cependant confronté à plusieurs possibilités quant à la manière d'exprimer son analyse. On voudrait trouver un mode d'expression qui soit simple à écrire et à manipuler, qui montre les caractéristiques importantes des besoins d'un algorithme au niveau des ressources, et qui élimine les détails fastidieux.

a) Analyse du tri par insertion

La durée d'exécution de la procédure TRI-INSERTION dépend de l'entrée : le tri d'un millier de nombres prend plus de temps que le tri de trois nombres. De plus, TRI-INSERTION peut demander des temps différents pour trier deux entrées de même taille, selon qu'elles sont déjà plus ou moins triées partiellement. En général, le temps d'exécution d'un algorithme croît avec la taille de l'entrée ; on a donc pris l'habitude d'exprimer le temps d'exécution d'un algorithme en fonction de la taille de son entrée. Reste à définir plus précisément ce que signifient « temps d'exécution » et « taille de l'entrée ».

Savoir ce que recouvre la notion de *taille de l'entrée*, cela dépend du problème étudié. Pour de nombreux problèmes, tels le tri ou le calcul de transformées de Fourier discrètes, la notion la plus naturelle est le *nombre d'éléments constituant l'entrée*, par exemple la longueur n du tableau à trier. Pour beaucoup d'autres problèmes, comme la multiplication de deux entiers, la meilleure mesure de la taille de l'entrée est le *nombre total de bits* nécessaire à la représentation de l'entrée dans la notation binaire habituelle. Parfois, il est plus approprié de décrire la taille de l'entrée avec deux nombres au lieu d'un seul. Par exemple, si l'entrée d'un algorithme est un graphe, on pourra décrire la taille de l'entrée par le nombre de sommets et le nombre d'arcs. Pour chaque problème, nous indiquerons la mesure utilisée pour exprimer la taille de l'entrée.

Le *temps d'exécution* d'un algorithme pour une entrée particulière est le nombre d'opérations élémentaires, ou « étapes », exécutées. Il est commode de définir la notion d'étape de façon qu'elle soit le plus indépendante possible de la machine. Pour le moment, nous adopterons le point de vue suivant. L'exécution de chaque ligne de pseudo code demande un temps constant. Deux lignes différentes peuvent prendre des temps différents, mais chaque exécution de la i -ème ligne prend un temps c_i , c_i étant une constante. Ce point de vue est compatible avec le modèle RAM et reflète la manière dont le pseudo code serait implémenté sur la plupart des ordinateurs réels.⁽⁴⁾

(4) Il faut noter ici quelques subtilités. Les étapes de calcul spécifiées en langage naturel sont souvent des variantes d'une procédure qui demande plus qu'un volume de temps constant. Par exemple, nous verrons plus loin dans ce livre que, quand nous disons « trier les points en fonction de leurs abscisses », cela demande en fait plus qu'un temps constant. Notez aussi qu'une instruction qui appelle une sous-routine demande un temps constant, alors même que le sous-programme, une fois invoqué, peut demander plus. Autrement dit, on

Dans l'étude qui suit, notre conception du temps d'exécution de TRI-INSERTION évoluera, pour remplacer une formule complexe utilisant tous les coûts d'instruction c_i par une notation bien plus simple, plus concise et plus facile à manipuler. Cette notation simplifiée nous aidera aussi à déterminer si un algorithme est plus performant qu'un autre.

Nous commencerons par présenter la procédure TRI-INSERTION avec le « coût » temporel de chaque instruction et le nombre de fois que chaque instruction est exécutée. Pour tout $j = 2, 3, \dots, n$, où $n = \text{longueur}[A]$, soit t_j le nombre de fois que le test de la boucle **tant que**, en ligne 5, est exécuté pour cette valeur de j . Quand une boucle **pour** ou **tant que** se termine normalement (c'est-à-dire, suite au test effectué dans l'en tête de la boucle), le test est exécuté une fois de plus que le corps de la boucle. On suppose que les commentaires ne sont pas des instructions exécutables et qu'ils consomment donc un temps nul.

TRI-INSERTION (A)	<i>coût</i>	<i>fois</i>
1 pour $j \leftarrow 2$ à $\text{longueur}[A]$	c_1	n
2 faire $\text{clé} \leftarrow A[j]$	c_2	$n - 1$
3 ▷ Insère $A[j]$ dans la suite triée $A[1 \dots j - 1]$.	0	$n - 1$
4 $i \leftarrow j - 1$	c_4	$n - 1$
5 tant que $i > 0$ et $A[i] > \text{clé}$	c_5	$\sum_{j=2}^n t_j$
6 faire $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i \leftarrow i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] \leftarrow \text{clé}$	c_8	$n - 1$

Le temps d'exécution de l'algorithme est la somme des temps d'exécution de chaque instruction exécutée; une instruction qui demande c_i étapes et qui est exécutée n fois compte pour $c_i n$ dans le temps d'exécution total⁽⁵⁾. Pour calculer $T(n)$, temps d'exécution de TRI-INSERTION, on additionne les produits des colonnes *coût* et *fois*, ce qui donne

$$\begin{aligned}
 T(n) = & c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\
 & + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1).
 \end{aligned}$$

Même pour des entrées ayant la même taille, le temps d'exécution d'un algorithme peut dépendre de l'entrée *particulière* ayant cette taille. Par exemple, dans

sépare le processus d'*appel* du sous-programme (passage des paramètres, etc.) du processus d'*exécution* du sous-programme.

(5) Cette caractéristique n'est pas forcément valable pour une ressource comme la mémoire. Une instruction qui référence m mots de mémoire et qui est exécutée n fois ne consomme pas nécessairement mn mots de mémoire au total.

TRI-INSERTION, le cas le plus favorable est celui où le tableau est déjà trié. Pour tout $j = 2, 3, \dots, n$, on trouve alors que $A[j] \leq \text{clé}$ en ligne 5 quand i a sa valeur initiale de $j - 1$. Donc $t_j = 1$ pour $j = 2, 3, \dots, n$, et le temps d'exécution associé au cas optimal est

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ce temps d'exécution peut être exprimé sous la forme $an + b$, a et b étant des constantes dépendant des coûts c_i des instructions ; c'est donc une **fonction linéaire** de n .

Si le tableau est trié dans l'ordre décroissant, alors on a le cas le plus défavorable. On doit comparer chaque élément $A[j]$ avec chaque élément du sous-tableau trié $A[1..j - 1]$, et donc $t_j = j$ pour $j = 2, 3, \dots, n$. Si l'on remarque que

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \quad \text{et} \quad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

(voir l'annexe A pour ce genre de sommation), on trouve que, dans le cas le plus défavorable, le temps d'exécution de TRI-INSERTION est

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n - 1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Le temps d'exécution du cas le plus défavorable s'exprime donc sous la forme $an^2 + bn + c$, a , b et c étant ici aussi des constantes qui dépendent des coûts c_i des instructions ; c'est donc une **fonction quadratique** de n .

Généralement, et c'est le cas du tri par insertion, le temps d'exécution d'un algorithme est constant pour une entrée donnée ; encore que, dans des chapitres ultérieurs, nous verrons quelques algorithmes « randomisés » intéressants dont le comportement peut varier même pour une entrée fixée.

b) Analyse du cas le plus défavorable et du cas moyen

Dans notre analyse du tri par insertion, nous nous sommes intéressés aussi bien au cas le plus favorable, celui où le tableau en entrée est déjà trié, qu'au cas le plus défavorable, celui où le tableau en entrée est trié en sens inverse. Dans la suite de ce livre, cependant, nous aurons pour objectif général de déterminer le **temps d'exécution du cas le plus défavorable**, c'est-à-dire le temps d'exécution maximal pour une *quelconque* entrée de taille n . Voici trois arguments en ce sens.

- Le temps d'exécution associé au cas le plus défavorable est une borne supérieure du temps d'exécution associé à une entrée quelconque. Connaître cette valeur nous permettra donc d'avoir la certitude que l'algorithme ne mettra jamais plus de temps que cette limite. Ainsi, point besoin de faire de conjecture tarabiscotée sur le temps d'exécution en espérant ne jamais rencontrer un cas encore pire.
- Pour certains algorithmes, le cas le plus défavorable survient assez souvent. Par exemple, quand on recherche une information dans une base de données, si cette information n'existe pas dans la base le cas le plus défavorable se présentera souvent pour l'algorithme de recherche.
- Il n'est pas rare que le « cas moyen » soit presque aussi mauvais que le cas le plus défavorable. Supposons que l'on choisisse au hasard n nombres, auxquels on applique ensuite le tri par insertion. Combien de temps demande la détermination de l'emplacement d'insertion de $A[j]$ dans le sous-tableau $A[1..j-1]$? En moyenne, la moitié des éléments de $A[1..j-1]$ sont inférieurs à $A[j]$ et la moitié lui sont supérieurs. Donc, en moyenne, on teste la moitié du sous-tableau $A[1..j-1]$, auquel cas $t_j = j/2$. Si l'on calcule alors le temps d'exécution global associé au cas moyen, on voit que c'est une fonction quadratique de la taille de l'entrée, tout comme le temps d'exécution du cas le plus défavorable.

Dans certains cas particuliers, nous nous intéresserons au *temps d'exécution moyen*, ou temps d'exécution *attendu*, d'un algorithme. Au chapitre 5, nous verrons la technique de l'*analyse probabiliste* qui permet de déterminer le temps d'exécution attendu. Toutefois, une difficulté concernant l'analyse du cas moyen est de savoir ce qu'est une entrée « moyenne » pour un problème particulier. Souvent, nous supposerons que toutes les entrées ayant une taille donnée sont équiprobables. Cette hypothèse n'est pas toujours vérifiée dans la pratique, mais nous pourrions parfois employer un *algorithme randomisé* qui force des choix aléatoires afin de permettre l'utilisation d'une analyse probabiliste.

c) *Ordre de grandeur*

Nous avons utilisé des hypothèses simplificatrices pour faciliter notre analyse de la procédure TRI-INSERTION. D'abord, nous avons ignoré le coût réel de chaque instruction en employant les constantes c_i pour représenter ces coûts. Ensuite, nous avons observé que même ces constantes nous donnent plus de détails que nécessaire : le temps d'exécution du cas le plus défavorable est $an^2 + bn + c$, a , b et c étant des constantes qui dépendent des coûts c_i des instructions. Nous avons donc ignoré non seulement les coûts réels des instructions, mais aussi les coûts abstraits c_i .

Nous allons à présent utiliser une simplification supplémentaire. Ce qui nous intéresse vraiment, c'est le *taux de croissance*, ou *ordre de grandeur*, du temps d'exécution. On ne considérera donc que le terme dominant d'une formule (par exemple an^2), puisque les termes d'ordre inférieur sont moins significatifs pour n grand. On ignorera également le coefficient constant du terme dominant, puisque les facteurs constants sont moins importants que l'ordre de grandeur pour ce qui est de la détermination de

l'efficacité du calcul pour les entrées volumineuses. On écrira donc que le tri par insertion, par exemple, a dans le cas le plus défavorable un temps d'exécution de $\Theta(n^2)$ (prononcer « théta de n -deux »). Nous utiliserons la notation Θ de façon informelle dans ce chapitre ; elle sera définie de manière plus précise au chapitre 3.

On considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur. Compte tenu des facteurs constants et des termes d'ordre inférieur, cette évaluation peut être erronée pour des entrées de faible volume. En revanche, pour des entrées de taille assez grande, un algorithme en $\Theta(n^2)$, par exemple, s'exécute plus rapidement, dans le cas le plus défavorable, qu'un algorithme en $\Theta(n^3)$.

Exercices

2.2.1 Exprimer la fonction $n^3/1000 - 100n^2 - 100n + 3$ à l'aide de la notation Θ .

2.2.2 On considère le tri suivant de n nombres rangés dans un tableau A : on commence par trouver le plus petit élément de A et on le permute avec $A[1]$. On trouve ensuite le deuxième plus petit élément de A et on le permute avec $A[2]$. On continue de cette manière pour les $n - 1$ premiers éléments de A . Écrire du pseudo code pour cet algorithme, connu sous le nom de **tri par sélection**. Quel est l'invariant de boucle de cet algorithme ? Pourquoi suffit-il d'exécuter l'algorithme pour les $n - 1$ premiers éléments ? Donner les temps d'exécution associés au cas optimal et au cas le plus défavorable en utilisant la notation Θ .

2.2.3 On considère une fois de plus la recherche linéaire (voir exercice 2.1.3). Combien d'éléments de la séquence d'entrée doit-on tester en moyenne, si l'on suppose que l'élément recherché a une probabilité égale d'être l'un quelconque des éléments du tableau ? Et dans le cas le plus défavorable ? Quels sont les temps d'exécution du cas moyen et du cas le plus défavorable, exprimés avec la notation Θ ? Justifier les réponses.

2.2.4 Comment modifier la plupart des algorithmes pour qu'ils aient un bon temps d'exécution dans le cas le plus favorable ?

2.3 CONCEPTION DES ALGORITHMES

Il existe de nombreuses façons de concevoir un algorithme. Le tri par insertion utilise une approche **incrémentale** : après avoir trié le sous-tableau $A[1..j - 1]$, on insère l'élément $A[j]$ au bon emplacement pour produire le sous-tableau trié $A[1..j]$.

Cette section va présenter une autre approche de conception, baptisée « diviser pour régner ». Nous utiliserons cette technique pour créer un algorithme de tri dont le temps d'exécution du cas le plus défavorable sera très inférieur à celui du tri par insertion. L'un des avantages des algorithmes diviser-pour-régner est que leurs temps d'exécution sont souvent faciles à déterminer, *via* des techniques qui seront vues au chapitre 4.

2.3.1 Méthode diviser-pour-régner

Nombre d'algorithmes utiles sont d'essence indérécurtivité *réursive* : pour résoudre le problème, ils s'appellent eux-mêmes, de manière réursive, une ou plusieurs fois pour traiter des sous-problèmes très similaires. Ces algorithmes suivent généralement une approche *diviser pour régner* : ils séparent le problème en plusieurs sous-problèmes semblables au problème initial mais de taille moindre, résolvent les sous-problèmes de façon réursive, puis combinent toutes les solutions pour produire la solution du problème original.

Le paradigme diviser-pour-régner implique trois étapes à chaque niveau de la récurtivité :

Diviser : le problème en un certain nombre de sous-problèmes.

Régner : sur les sous-problèmes en les résolvant de manière réursive. Si la taille d'un sous-problème est suffisamment réduite, on peut toutefois le résoudre directement.

Combiner : les solutions des sous-problèmes pour produire la solution du problème original.

L'algorithme du *tri par fusion* suit fidèlement la méthodologie diviser-pour-régner. Intuitivement, il agit de la manière suivante :

Diviser : Diviser la suite de n éléments à trier en deux sous-suites de $n/2$ éléments chacune.

Régner : Trier les deux sous-suites de manière réursive en utilisant le tri par fusion.

Combiner : Fusionner les deux sous-suites triées pour produire la réponse triée.

La récurtivité s'arrête quand la séquence à trier a une longueur 1, auquel cas il n'y a plus rien à faire puisqu'une suite de longueur 1 est déjà triée.

La clé de voûte de l'algorithme du tri par fusion, c'est la fusion de deux séquences triées dans l'étape « combiner ». Pour faire cette fusion, nous utilisons une procédure auxiliaire $FUSION(A, p, q, r)$, A étant un tableau et p , q et r étant des indices numérotant des éléments du tableau tels que $p \leq q < r$. La procédure suppose que les sous-tableaux $A[p \dots q]$ et $A[q + 1 \dots r]$ sont triés. Elle les *fusionne* pour en faire un même sous-tableau trié, qui remplace le sous-tableau courant $A[p \dots r]$.

Notre procédure $FUSION$ a une durée $\Theta(n)$, où $n = r - p + 1$ est le nombre d'éléments fusionnés. Voici comment elle fonctionne. En reprenant l'exemple du jeu de cartes, supposez que l'on ait deux piles de cartes posées à l'endroit sur la table. Chaque pile est triée de façon à ce que la carte la plus faible soit en haut. On veut fusionner les deux piles pour obtenir une pile unique triée, dans laquelle les cartes seront à l'envers. L'étape fondamentale consiste à choisir la plus faible des deux cartes occupant les sommets respectifs des deux piles, à la retirer de sa pile (ce qui a pour effet d'exposer une nouvelle carte), puis à la placer à l'envers sur la pile résultante. On répète cette étape jusqu'à épuisement de l'une des piles de départ, après quoi il suffit de prendre la pile qui reste et de la placer à l'envers sur la pile résultante. Au

niveau du calcul, chaque étape fondamentale prend un temps constant, vu que l'on se contente de comparer les deux cartes du haut. Comme l'on effectue au plus n étapes fondamentales, la fusion prend une durée $\Theta(n)$.

Le pseudo code suivant implémente la méthode que nous venons d'exposer, mais avec une astuce supplémentaire qui évite de devoir, à chaque étape fondamentale, vérifier si l'une des piles est vide. L'idée est de placer en bas de chaque pile une carte *sentinelle* contenant une valeur spéciale qui nous permet de simplifier le code. Ici, nous utiliserons ∞ comme valeur sentinelle ; ainsi, chaque fois qu'il y a apparition d'une carte portant la valeur ∞ , elle ne peut pas être la carte la plus faible sauf si les deux piles exposent en même temps leurs cartes sentinelle. Mais alors, c'est que toutes les autres cartes ont déjà été placées sur la pile de sortie. Comme nous savons à l'avance qu'il y aura exactement $r - p + 1$ cartes sur la pile de sortie, nous pourrons arrêter lorsque nous aurons effectué ce nombre d'étapes.

```

FUSION( $A, p, q, r$ )
1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  créer tableaux  $L[1 \dots n_1 + 1]$  et  $R[1 \dots n_2 + 1]$ 
4  pour  $i \leftarrow 1$  à  $n_1$ 
5      faire  $L[i] \leftarrow A[p + i - 1]$ 
6  pour  $j \leftarrow 1$  à  $n_2$ 
7      faire  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 pour  $k \leftarrow p$  à  $r$ 
13     faire si  $L[i] \leq R[j]$ 
14         alors  $A[k] \leftarrow L[i]$ 
15              $i \leftarrow i + 1$ 
16     sinon  $A[k] \leftarrow R[j]$ 
17          $j \leftarrow j + 1$ 

```

Voici le détail du fonctionnement de la procédure FUSION. La ligne 1 calcule la longueur n_1 du sous-tableau $A[p..q]$ et la ligne 2 calcule la longueur n_2 du sous-tableau $A[q + 1..r]$. On crée des tableaux L et R de longueurs $n_1 + 1$ et $n_2 + 1$, respectivement, en ligne 3. La boucle **pour** des lignes 4–5 copie le sous-tableau $A[p..q]$ dans $L[1..n_1]$ et la boucle **pour** des lignes 6–7 copie le sous-tableau $A[q + 1..r]$ dans $R[1..n_2]$. Les lignes 8–9 placent les sentinelles aux extrémités des tableaux L et R . Les lignes 10–17, illustrées sur la figure 2.3, effectuent les $r - p + 1$ étapes fondamentales en conservant l'invariant de boucle que voici :

Au début de chaque itération de la boucle **pour** des lignes 12–17, le sous-tableau $A[p..k - 1]$ contient les $k - p$ plus petits éléments de $L[1..n_1 + 1]$ et

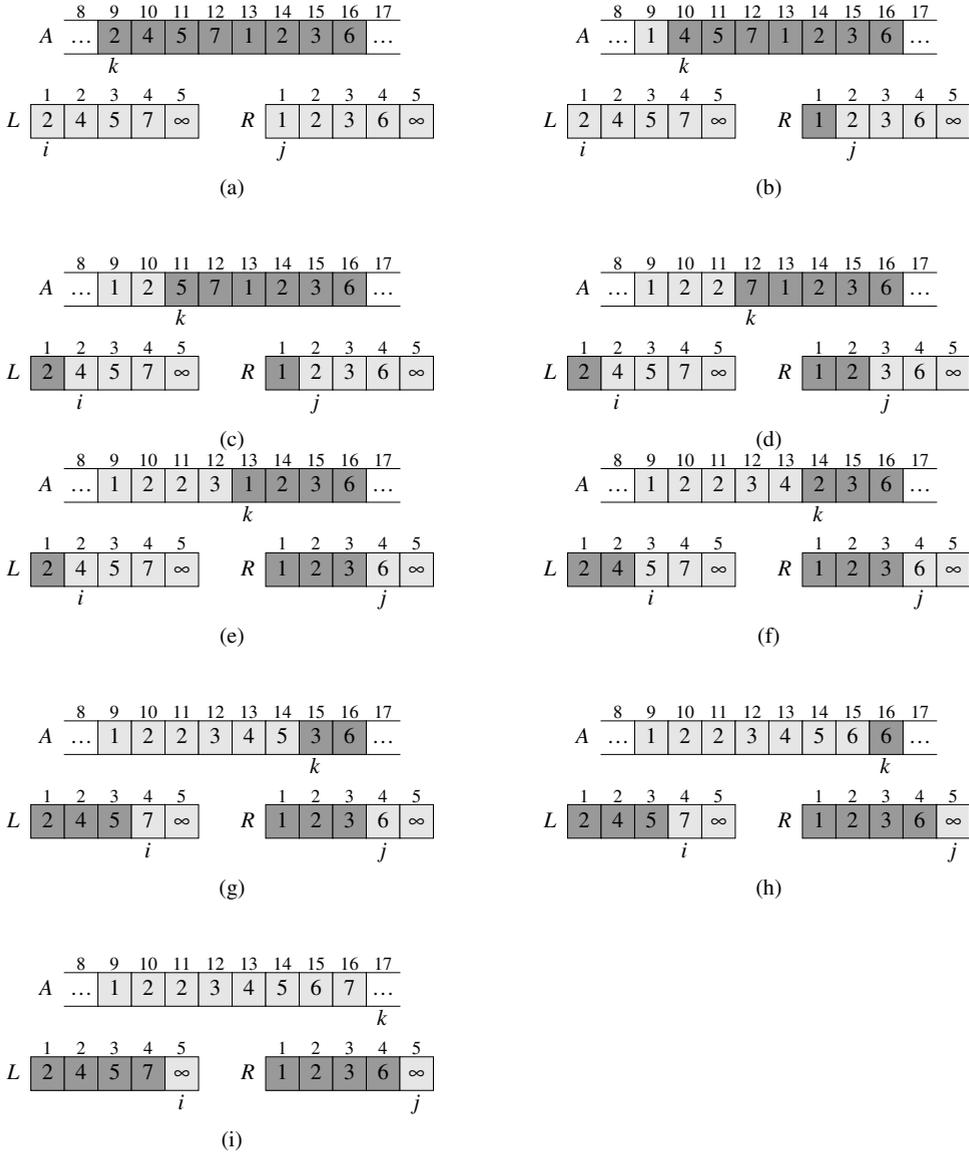


Figure 2.3 Le fonctionnement des lignes 10–17 dans l’appel `FUSION(A, 9, 12, 16)`, quand le sous-tableau $A[9..16]$ contient la séquence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. Après copie et insertion des sentinelles, le tableau L contient $\langle 2, 4, 5, 7, \infty \rangle$ et le tableau R contient $\langle 1, 2, 3, 6, \infty \rangle$. Les cases en gris clair dans A contiennent leurs valeurs définitives, alors que les cases en gris clair dans L et R contiennent des valeurs qui n’ont pas encore été copiées dans A . Prises ensemble, les cases en gris clair contiennent à tout moment les valeurs qui étaient à l’origine dans $A[9..16]$, avec en plus les deux sentinelles. Les cases en gris foncé dans A contiennent des valeurs qui seront remplacées par des duplications ultérieures, alors que les cases en gris foncé dans L et R contiennent des valeurs qui ont été déjà copiées dans A . (a)–(h) Les tableaux A , L et R , avec leurs indices respectifs k , i et j avant chaque itération de la boucle des lignes 12–17. (i) Les tableaux et les indices à la fin. À ce stade, le sous-tableau dans $A[9..16]$ est trié et les deux sentinelles dans L et R sont les seuls éléments de ces tableaux qui n’ont pas été copiés dans A .

$R[1 \dots n_2 + 1]$, en ordre trié. En outre, $L[i]$ et $R[j]$ sont les plus petits éléments de leurs tableaux à ne pas avoir été copiés dans A .

Il faut montrer que cet invariant est vrai avant la première itération de la boucle **pour** des lignes 12–17, que chaque itération de la boucle conserve l'invariant, et enfin que l'invariant fournit une propriété utile pour prouver la conformité de la procédure quand la boucle se termine.

Initialisation : Avant la première itération de la boucle, on a $k = p$, de sorte que le sous-tableau $A[p \dots k - 1]$ est vide. Ce sous-tableau vide contient les $k - p = 0$ plus petits éléments de L et R ; et, comme $i = j = 1$, $L[i]$ et $R[j]$ sont les plus petits éléments de leurs tableaux à ne pas avoir été copiés dans A .

Conservation : Pour montrer que chaque itération conserve l'invariant, supposons en premier lieu que $L[i] \leq R[j]$. Alors $L[i]$ est le plus petit élément qui n'a pas encore été copié dans A . Comme $A[p \dots k - 1]$ contient les $k - p$ plus petits éléments, après que la ligne 14 a copié $L[i]$ dans $A[k]$, le sous-tableau $A[p \dots k]$ contient les $k - p + 1$ plus petits éléments. Incrémenter k (dans l'actualisation de la boucle **pour**) et i (en ligne 15) recrée l'invariant pour l'itération suivante. Si l'on a $L[i] > R[j]$, alors les lignes 16–17 font l'action idoine pour conserver l'invariant.

Terminaison : À la fin de la boucle, $k = r + 1$. D'après l'invariant, le sous-tableau $A[p \dots k - 1]$, qui est $A[p \dots r]$, contient les $k - p = r - p + 1$ plus petits éléments de $L[1 \dots n_1 + 1]$ et $R[1 \dots n_2 + 1]$, dans l'ordre trié. Les tableaux L et R , à eux deux, contiennent $n_1 + n_2 + 2 = r - p + 3$ éléments. Tous les éléments, sauf les deux plus forts, ont été copiés dans A , et ces deux plus gros éléments ne sont autres que les sentinelles.

Pour voir que la procédure FUSION s'exécute en $\Theta(n)$ temps, avec $n = r - p + 1$, observez que chacune des lignes 1–3 et 8–11 prend un temps constant, que les boucles **pour** des lignes 4–7 prennent $\Theta(n_1 + n_2) = \Theta(n)$ temps, ⁽⁶⁾ et qu'il y a n itérations de la boucle **pour** des lignes 12–17, chacune d'elles prenant un temps constant.

Nous pouvons maintenant employer la procédure FUSION comme sous-routine de l'algorithme du tri par fusion. La procédure TRI-FUSION(A, p, r) trie les éléments du sous-tableau $A[p \dots r]$. Si $p \geq r$, le sous-tableau a au plus un seul élément et il est donc déjà trié. Sinon, l'étape diviser se contente de calculer un indice q qui partitionne $A[p \dots r]$ en deux sous-tableaux : $A[p \dots q]$, contenant $\lceil n/2 \rceil$ éléments, et $A[q + 1 \dots r]$, contenant $\lfloor n/2 \rfloor$ éléments. ⁽⁷⁾

(6) Nous verrons au chapitre 3 comment interpréter de manière formelle les équations contenant la notation Θ .

(7) L'expression $\lceil x \rceil$ désigne le plus petit entier supérieur ou égal à x , alors que $\lfloor x \rfloor$ désigne le plus grand entier inférieur ou égal à x . Ces notations sont définies au chapitre 3. Le moyen le plus simple de vérifier que donner à q la valeur $\lfloor (p + r)/2 \rfloor$ produit des sous-tableaux $A[p \dots q]$ et $A[q + 1 \dots r]$ de tailles respectives $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, c'est d'examiner les quatre cas possibles selon que chacune des valeurs p et r est impaire ou paire.

```

TRI-FUSION( $A, p, r$ )
1  si  $p < r$ 
2    alors  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
3      TRI-FUSION( $A, p, q$ )
4      TRI-FUSION( $A, q+1, r$ )
5      FUSION( $A, p, q, r$ )

```

Pour trier toute la séquence $A = \langle A[1], A[2], \dots, A[n] \rangle$, on fait l'appel initial TRI-FUSION($A, 1, \text{longueur}[A]$) (ici aussi, $\text{longueur}[A] = n$). La figure 2.4 illustre le fonctionnement de la procédure, du bas vers le haut, quand n est une puissance de 2. L'algorithme consiste à fusionner des paires de séquences à 1 élément pour former des séquences triées de longueur 2, à fusionner des paires de séquences de longueur 2 pour former des séquences triées de longueur 4, etc. jusqu'à qu'il y ait fusion de deux séquences de longueur $n/2$ pour former la séquence triée définitive de longueur n .

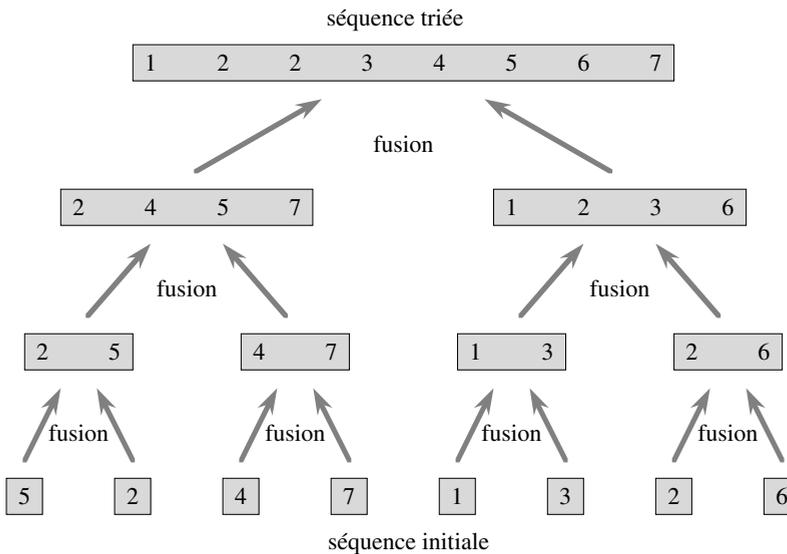


Figure 2.4 Le fonctionnement du tri par fusion sur le tableau $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Les longueurs des séquences triées en cours de fusion augmentent à mesure que l'algorithme remonte du bas vers le haut.

2.3.2 Analyse des algorithmes diviser-pour-régner

Lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une *équation de récurrence*, ou *récurrence*, qui décrit le temps d'exécution global pour un problème de taille n à partir du temps d'exécution pour des entrées de taille moindre. On peut alors se servir d'outils mathématiques

pour résoudre la récurrence et trouver des bornes pour les performances de l'algorithme.

Une récurrence pour le temps d'exécution d'un algorithme diviser-pour-régner s'appuie sur les trois étapes du paradigme de base. Comme précédemment, soit $T(n)$ le temps d'exécution d'un problème de taille n . Si la taille du problème est suffisamment petite, disons $n \leq c$ pour une certaine constante c , la solution directe prend un temps constant que l'on écrit $\Theta(1)$. Supposons que l'on divise le problème en a sous-problèmes, la taille de chacun étant $1/b$ de la taille du problème initial. (Pour le tri par fusion, tant a que b valent 2, mais nous verrons beaucoup d'algorithmes diviser-pour-régner dans lesquels $a \neq b$.) Si l'on prend un temps $D(n)$ pour diviser le problème en sous-problèmes et un temps $C(n)$ pour construire la solution finale à partir des solutions aux sous-problèmes, on obtient la récurrence

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon.} \end{cases}$$

Au chapitre 4, nous verrons comment résoudre des récurrences communes ayant cette forme.

a) Analyse du tri par fusion

Bien que le pseudo-code de TRI-FUSION s'exécute correctement quand le nombre d'éléments n'est pas pair, notre analyse fondée sur la récurrence sera simplifiée si nous supposons que la taille du problème initial est une puissance de deux. Chaque étape diviser génère alors deux sous-séquences de taille $n/2$ exactement. Au chapitre 4, nous verrons que cette supposition n'affecte pas l'ordre de grandeur de la solution de la récurrence.

Nous raisonnerons comme suit pour mettre en œuvre la récurrence pour $T(n)$, temps d'exécution du cas le plus défavorable du tri par fusion de n nombres. Le tri par fusion d'un seul élément prend un temps constant ; avec $n > 1$ éléments, on segmente le temps d'exécution de la manière suivante.

Diviser : L'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant. Donc $D(n) = \Theta(1)$.

Régner : On résout récursivement deux sous-problèmes, chacun ayant la taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution.

Combiner : Nous avons déjà noté que la procédure FUSION sur un sous-tableau à n éléments prenait un temps $\Theta(n)$, de sorte que $C(n) = \Theta(n)$.

Quand on ajoute les fonctions $D(n)$ et $C(n)$ pour l'analyse du tri par fusion, on ajoute une fonction qui est $\Theta(n)$ et une fonction qui est $\Theta(1)$. Cette somme est une fonction linéaire de n , à savoir $\Theta(n)$. L'ajouter au terme $2T(n/2)$ de l'étape régner donne la récurrence pour $T(n)$, temps d'exécution du tri par fusion dans le cas le plus défavorable :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(n/2) + \Theta(n) & \text{si } n > 1. \end{cases} \quad (2.1)$$

Au chapitre 4, nous verrons le « théorème général » grâce auquel on peut montrer que $T(n)$ est $\Theta(n \lg n)$, où $\lg n$ désigne $\log_2 n$. Comme la fonction logarithme croît plus lentement que n'importe quelle fonction linéaire, pour des entrées suffisamment grandes, le tri par fusion, avec son temps d'exécution $\Theta(n \lg n)$, est plus efficace que le tri par insertion dont le temps d'exécution vaut $\Theta(n^2)$ dans le cas le plus défavorable.

On n'a pas besoin du théorème général pour comprendre intuitivement pourquoi la solution de la récurrence (2.1) est $T(n) = \Theta(n \lg n)$. Réécrivons la récurrence (2.1) comme suit

$$T(n) = \begin{cases} c & \text{si } n = 1, \\ 2T(n/2) + cn & \text{si } n > 1, \end{cases} \quad (2.2)$$

la constante c représentant le temps requis pour résoudre des problèmes de taille 1, ainsi que le temps par élément de tableau des étapes diviser et combiner. ⁽⁸⁾

La figure 2.5 montre comment on peut résoudre la récurrence (2.2). Pour des raisons de commodité, on suppose que n est une puissance exacte de 2. La partie (a) de la figure montre $T(n)$ qui, en partie (b), a été développé pour devenir un arbre équivalent représentant la récurrence. Le terme cn est la racine (le coût au niveau supérieur de récursivité), et les deux sous-arbres de la racine sont les deux récurrences plus petites $T(n/2)$. La partie (c) montre le processus une étape plus loin, après expansion de $T(n/2)$. Le coût de chacun des deux sous-nœuds, au deuxième niveau de récursivité, est $cn/2$. On continue de développer chaque nœud de l'arbre en le divisant en ses parties constituantes telles que déterminées par la récurrence, jusqu'à ce que les tailles de problème tombent à 1, chacune ayant alors un coût c . La partie (d) montre l'arbre résultant.

Ensuite, on cumule les coûts sur chaque niveau. Le niveau supérieur a un coût total cn , le niveau suivant a un coût total $c(n/2) + c(n/2) = cn$, le niveau suivant a un coût total $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, etc. Plus généralement le niveau i au-dessous de la racine a 2^i nœuds, dont chacun contribue pour un coût $c(n/2^i)$, de sorte que ce i -ème niveau a un coût total $2^i c(n/2^i) = cn$. Au niveau le plus bas, il y a n nœuds dont chacun contribue pour un coût de c , ce qui donne un coût total de cn .

Le nombre total de niveaux de cet « arbre de récursivité », tel qu'indiqué sur la figure 2.5, est $\lg n + 1$. Ce fait se laisse facilement vérifier par un raisonnement inductif informel. Le cas de base se produit pour $n = 1$, auquel cas il n'y a qu'un seul niveau. Comme $\lg 1 = 0$, on voit que $\lg n + 1$ donne le nombre correct de niveaux. Supposons maintenant, comme hypothèse de récurrence, que le nombre de niveaux d'un arbre de récursivité à 2^i nœuds soit $\lg 2^i + 1 = i + 1$ (car, pour une valeur quelconque de i , on a $\lg 2^i = i$). Comme on suppose que la taille originelle de l'entrée est une puissance de 2, la taille à considérer pour l'entrée suivante est 2^{i+1} . Un arbre à 2^{i+1}

(8) Il est très improbable que la même constante représente exactement et le temps de résolution des problèmes de taille 1 et le temps par élément de tableau des étapes diviser et combiner. On peut contourner cette difficulté en prenant pour c le plus grand de ces temps et en supposant que notre récurrence donne une borne supérieure du temps d'exécution, ou bien en prenant pour c le plus petit de ces temps et en partant du principe que notre récurrence donne une borne inférieure du temps d'exécution. Les deux bornes sont de l'ordre de $n \lg n$ et, à elles deux, donnent un temps d'exécution en $\Theta(n \lg n)$.

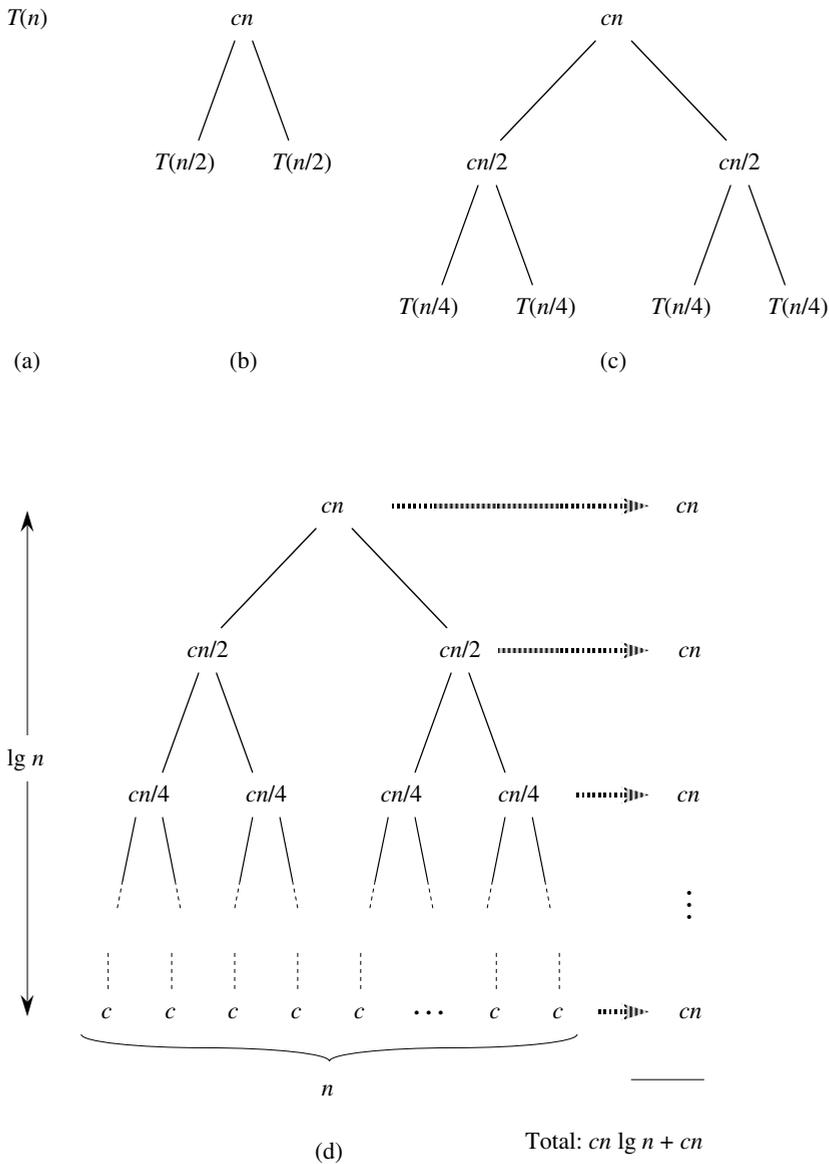


Figure 2.5 La construction d'un arbre de récursivité pour la récurrence $T(n) = 2T(n/2) + cn$. La partie (a) montre $T(n)$, progressivement développé en (b)–(d) pour former l'arbre de récursivité. L'arbre complet, en partie (d), a $\lg n + 1$ niveaux (c'est-à-dire, il a une hauteur $\lg n$ comme indiquée), sachant que chaque niveau contribue pour un coût total de cn . Le coût global est donc $cn \lg n + cn$, c'est à dire $\Theta(n \lg n)$.

nœuds ayant un niveau de plus qu'un arbre à 2^i nœuds, le nombre total de niveaux est donc $(i + 1) + 1 = \lg 2^{i+1} + 1$.

Pour calculer le coût total représenté par la récurrence (2.2), il suffit d'additionner les coûts de tous les niveaux. Il y a $\lg n + 1$ niveaux, chacun coûtant cn , ce qui donne un coût global de $cn(\lg n + 1) = cn \lg n + cn$. En ignorant le terme d'ordre inférieur et la constante c , on arrive au résultat souhaité qui est $\Theta(n \lg n)$.

Exercices

2.3.1 En s'inspirant de la figure 2.4, illustrer le fonctionnement du tri par fusion sur le tableau $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3.2 Réécrire la procédure FUSION de telle sorte qu'elle n'emploie pas de sentinelles mais qu'à la place elle s'arrête quand l'un des deux tableaux L et R a eu tous ses éléments copiés dans A , en copiant alors le reste de l'autre tableau dans A .

2.3.3 Utiliser l'induction mathématique pour montrer que, lorsque n est une puissance exacte de 2, la solution de la récurrence

$$T(n) = \begin{cases} 2 & \text{si } n = 2, \\ 2T(n/2) + n & \text{si } n = 2^k, \text{ pour } k > 1 \end{cases}$$

est $T(n) = n \lg n$.

2.3.4 Le tri par insertion peut être exprimé sous la forme d'une procédure récursive de la manière suivante. Pour trier $A[1..n]$, on trie récursivement $A[1..n-1]$ puis on insère $A[n]$ dans le tableau trié $A[1..n-1]$. Écrire une récurrence pour le temps d'exécution de cette version récursive du tri par insertion.

2.3.5 En reprenant le problème de la recherche (voir exercice 2.1.3), observez que, si la séquence A est triée, on peut comparer le milieu de la séquence avec v et supprimer la moitié de la séquence pour la suite des opérations. La **recherche dichotomique** est un algorithme qui répète cette procédure, en divisant par deux à chaque fois la taille de la partie restante de la séquence. Écrire le pseudo code, itératif ou récursif, de la recherche dichotomique. Expliquer pourquoi le temps d'exécution de la recherche dichotomique, dans le cas le plus défavorable, est $\Theta(\lg n)$.

2.3.6 On observe que la boucle **tant que** des lignes 5 – 7 de la procédure TRI-INSERTION de la section 2.1 utilise une recherche linéaire pour parcourir (à rebours) le sous-tableau trié $A[1..j-1]$. Est-il possible d'utiliser à la place une recherche dichotomique (voir l'exercice 2.3.7) pour améliorer le temps d'exécution global du tri par insertion, dans le cas le plus défavorable, de façon qu'il devienne $\Theta(n \lg n)$?

2.3.7 ★ Décrire un algorithme en $\Theta(n \lg n)$ qui, étant donnés un ensemble S de n entiers et un autre entier x , détermine s'il existe ou non deux éléments de S dont la somme vaut exactement x .

PROBLÈMES

2.1. Tri par insertion sur petits tableaux dans le cadre du tri par fusion

Bien que, dans le pire des cas, le tri par fusion s'exécute en $\Theta(n \lg n)$ et le tri par insertion en $\Theta(n^2)$, les facteurs constants du tri par insertion le rendent plus rapide pour n petit. Il est donc logique d'utiliser le tri par insertion à l'intérieur du tri par fusion lorsque les sous-problèmes deviennent suffisamment petits. On va étudier la modification suivante du tri par fusion : n/k sous-listes de longueur k sont triées via tri par insertion, puis fusionnées à l'aide du mécanisme de fusion classique. k est une valeur à déterminer.

- a. Montrer que les n/k sous-listes, chacune de longueur k , peuvent être triées via tri par insertion avec un temps $\Theta(nk)$ dans le cas le plus défavorable.
- b. Montrer que les sous-listes peuvent être fusionnées en $\Theta(n \lg(n/k))$ dans le cas le plus défavorable.
- c. Sachant que l'algorithme modifié s'exécute en $\Theta(nk + n \lg(n/k))$ dans le cas le plus défavorable, quelle est la plus grande valeur asymptotique (notation Θ) de k , en tant que fonction de n , pour laquelle l'algorithme modifié a le même temps d'exécution asymptotique que le tri par fusion classique ?
- d. Comment doit-on choisir k en pratique ?

2.2. Conformité du tri à bulles

Le tri à bulles est un algorithme de tri très populaire. Son fonctionnement s'appuie sur des permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre.

TRI-BULLES(A)

```

1  pour  $i \leftarrow 1$  à longueur[ $A$ ]
2      faire pour  $j \leftarrow$  length[ $A$ ] decr jusqu'à  $i + 1$ 
3          faire si  $A[j] < A[j - 1]$ 
4              alors permuter  $A[j] \leftrightarrow A[j - 1]$ 
```

- a. Soit A' le résultat de TRI-BULLES(A). Pour prouver la conformité de TRI-BULLES, il faut montrer qu'il se termine et que

$$A'[1] \leq A'[2] \leq \dots \leq A'[n], \quad (2.3)$$

avec $n = \text{longueur}[A]$. Que faut-il prouver d'autre pour démontrer que TRI-BULLES trie réellement ?

Les deux parties suivantes vont montrer l'inégalité (2.3).

- b. Définir de manière précise un invariant pour la boucle **pour** des lignes 2–4, puis montrer que cet invariant est vérifié. La démonstration doit employer la structure d'invariant de boucle exposée dans ce chapitre.

- c. En utilisant la condition de finalisation de l'invariant démontrée en (b), définir un invariant pour la boucle **pour** des lignes 1–4 qui permette de prouver l'inégalité (2.3). La démonstration doit employer la structure d'invariant de boucle exposée dans ce chapitre.
- d. Quel est le temps d'exécution du tri à bulles dans le cas le plus défavorable ? Quelles sont ses performances, comparées au temps d'exécution du tri par insertion ?

2.3. Conformité de la règle de Horner

Le code suivant implémente la règle de Horner relative à l'évaluation d'un polynôme

$$P(x) = \sum_{k=0}^n a_k x^k = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n) \cdots)),$$

étant donnés les coefficients a_0, a_1, \dots, a_n et une valeur de x :

```

1  y ← 0
2  i ← n
3  tant que i ≥ 0
4      faire y ← ai + x · y
5      i ← i - 1
```

- a. Quel est le temps d'exécution asymptotique de ce fragment de code ?
- b. Écrire du pseudo code qui implémente l'algorithme naïf d'évaluation polynomiale, lequel calcule chaque terme du polynôme ex nihilo. Quel est le temps d'exécution de cet algorithme ? Quelles sont ses performances, comparées à celles de la règle de Horner ?
- c. Prouver que ce qui suit est un invariant pour la boucle **tant que** des lignes 3–5. Au début de chaque itération de la boucle **tant que** des lignes 3–5,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$$

On considérera qu'une sommation sans termes est égale à 0. La démonstration doit se conformer à la structure d'invariant de boucle exposée dans ce chapitre et montrer que, à la fin, on a $y = \sum_{k=0}^n a_k x^k$.

- d. Conclure en démontrant que le fragment de code donné évalue correctement un polynôme caractérisé par les coefficients a_0, a_1, \dots, a_n .

2.4. Inversions

Soit $A[1..n]$ un tableau de n nombres distincts. Si $i < j$ et $A[i] > A[j]$, on dit que le couple (i, j) est une *inversion* de A .

- a. Donner les cinq inversions du tableau $\langle 2, 3, 8, 6, 1 \rangle$.

- b. Quel est le tableau dont les éléments appartiennent à l'ensemble $\{1, 2, \dots, n\}$ qui a le plus d'inversions ? Combien en possède-t-il ?
- c. Quelle est la relation entre le temps d'exécution du tri par insertion et le nombre d'inversion du tableau en entrée ? Justifier la réponse.
- d. Donner un algorithme qui détermine en un temps $\Theta(n \lg n)$, dans le cas le plus défavorable, le nombre d'inversions présentes dans une permutation quelconque de n éléments. (*Conseil* : Modifier le tri par fusion.)

NOTES

En 1968, Knuth publia le premier des trois volumes ayant le titre général *The Art of Computer Programming* [182, 183, 185]. Le premier volume introduisait l'étude moderne des algorithmes informatiques en insistant sur l'analyse du temps d'exécution, et la série complète reste une référence attrayante et utile pour nombre de sujets présentés ici. Selon Knuth, le mot « algorithme » vient de « al-Khowârizmî » qui était un mathématicien Persan du neuvième siècle.

Aho, Hopcroft et Ullman [5] ont plaidé en faveur de l'analyse asymptotique des algorithmes comme moyen de comparer les performances relatives. Ils ont également popularisé l'utilisation des relations de récurrence pour décrire les temps d'exécution des algorithmes récursifs.

Knuth [185] offre une étude encyclopédique de nombreux algorithmes de tri. Sa comparaison des algorithmes de tri (page 381) contient des analyses dénombrant très exactement les étapes, comme celle que nous avons effectuée ici pour le tri par insertion. L'étude par Knuth du tri par insertion englobe plusieurs variantes de l'algorithme. La plus importante est le tri de Shell, dû à D. L. Shell, qui utilise le tri par insertion sur des sous-séquences périodiques de l'entrée pour produire un algorithme de tri plus rapide.

Knuth décrit également le tri par fusion. Il mentionne qu'une machine mécanique capable de fusionner deux jeux de cartes perforées en une seule passe fut inventée en 1938. Il semblerait que J. von Neumann, l'un des pionniers de l'informatique, ait écrit un programme de tri par fusion pour l'ordinateur EDVAC en 1945.

Gries [133] contient l'historique des essais de démonstration de la conformité des programmes ; il attribue à P. Naur le premier article paru sur le sujet. Gries attribue la paternité des invariants de boucle à R. W. Floyd. Le manuel de Mitchell [222] présente des travaux plus récents en matière de démonstration de la justesse des programmes.

Chapitre 3

Croissance des fonctions

L'ordre de grandeur du temps d'exécution d'un algorithme, défini au chapitre 2, donne une caractérisation simple de l'efficacité de l'algorithme et permet également de comparer les performances relatives d'algorithmes servant à faire le même travail. Quand la taille de l'entrée n devient suffisamment grande, le tri par fusion, avec son temps d'exécution en $\Theta(n \lg n)$ pour le cas le plus défavorable, l'emporte sur le tri par insertion, dont le temps d'exécution dans le cas le plus défavorable est en $\Theta(n^2)$. Bien qu'il soit parfois possible de déterminer le temps d'exécution exact d'un algorithme, comme nous l'avons fait pour le tri par insertion au chapitre 2, cette précision supplémentaire ne vaut généralement pas la peine d'être calculée. Pour des entrées suffisamment grandes, les effets des constantes multiplicatives et des termes d'ordre inférieur d'un temps d'exécution exact sont négligeables par rapport aux effets de la taille de l'entrée.

Quand on prend des entrées suffisamment grandes pour que seul compte réellement l'ordre de grandeur du temps d'exécution, on étudie les performances *asymptotiques* des algorithmes. En clair, on étudie la façon dont augmente *à la limite* le temps d'exécution d'un algorithme quand la taille de l'entrée augmente indéfiniment. En général, un algorithme plus efficace asymptotiquement qu'un autre constitue le meilleur choix, sauf pour les entrées très petites.

Ce chapitre va présenter plusieurs méthodes classiques pour la simplification de l'analyse asymptotique des algorithmes. La première section définira plusieurs types de « notation asymptotique », dont nous avons déjà vu un exemple avec la notation Θ . Seront ensuite présentées diverses conventions de notation utilisées tout au long de ce livre. Le chapitre se terminera par une révision du comportement de fonctions qui reviennent fréquemment dans l'analyse des algorithmes.

3.1 NOTATION ASYMPTOTIQUE

Les notations que nous utiliserons pour décrire le temps d'exécution asymptotique d'un algorithme sont définies en termes de fonctions dont le domaine de définition est l'ensemble des entiers naturels $\mathbf{N} = \{0, 1, 2, \dots\}$. De telles notations sont pratiques pour décrire la fonction $T(n)$ donnant le temps d'exécution du cas le plus défavorable, qui n'est généralement définie que sur des entrées de taille entière. Cependant, il est parfois avantageux d'étendre *abusivement* la notation asymptotique de diverses manières. Par exemple, la notation peut être facilement étendue au domaine des nombres réels, ou, inversement, réduite à un sous-ensemble des entiers naturels. Il est donc important de comprendre la signification précise de la notation, de sorte que, même si l'on en abuse, on n'en *mésuse* pas. Cette section va définir les notations asymptotiques fondamentales et présenter certains abus fréquents.

a) Notation Θ

Au chapitre 2, nous avons trouvé que le temps d'exécution, dans le cas le plus défavorable, du tri par insertion est $T(n) = \Theta(n^2)$. Définissons le sens de cette notation. Pour une fonction donnée $g(n)$, on note $\Theta(g(n))$ l'ensemble de fonctions suivant :

$$\Theta(g(n)) = \{f(n) : \text{il existe des constantes positives } c_1, c_2 \text{ et } n_0 \text{ telles que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ pour tout } n \geq n_0\} .^{(1)}$$

Une fonction $f(n)$ appartient à l'ensemble $\Theta(g(n))$ s'il existe des constantes positives c_1 et c_2 telles que $f(n)$ puisse être « prise en sandwich » entre $c_1 g(n)$ et $c_2 g(n)$, pour n assez grand. Comme $\Theta(g(n))$ est un ensemble, on pourrait écrire « $f(n) \in \Theta(g(n))$ » pour indiquer que $f(n)$ est un membre de $\Theta(g(n))$. En fait, on préfère écrire « $f(n) = \Theta(g(n))$ ». Cet emploi abusif de l'égalité pour représenter l'appartenance à un ensemble peut dérouter au début, mais nous verrons plus loin dans cette section que cela offre des avantages.

La figure 3.1(a) donne une représentation intuitive de fonctions $f(n)$ et $g(n)$ vérifiant $f(n) = \Theta(g(n))$. Pour toutes les valeurs de n situées à droite de n_0 , la valeur de $f(n)$ est supérieure ou égale à $c_1 g(n)$ et inférieure ou égale à $c_2 g(n)$. Autrement dit, pour tout $n \geq n_0$, la fonction $f(n)$ est égale à $g(n)$ à un facteur constant près. On dit que $g(n)$ est une *borne asymptotiquement approchée* de $f(n)$.

La définition de $\Theta(g(n))$ impose que chaque membre $f(n) \in \Theta(g(n))$ soit *asymptotiquement positif*, c'est-à-dire que $f(n)$ soit toujours positive pour n suffisamment grand. (Une fonction *asymptotiquement strictement positive* est une fonction qui est positive pour n suffisamment grand.) En conséquence, la fonction $g(n)$ elle-même doit être asymptotiquement positive ; sinon, l'ensemble $\Theta(g(n))$ serait vide. On supposera donc que toute fonction utilisée dans la notation Θ est asymptotiquement positive. Cette hypothèse reste valable pour toutes les autres notations asymptotiques définies dans ce chapitre.

(1) Dans la notation ensembliste, le caractère « : » est synonyme de « tel que ».

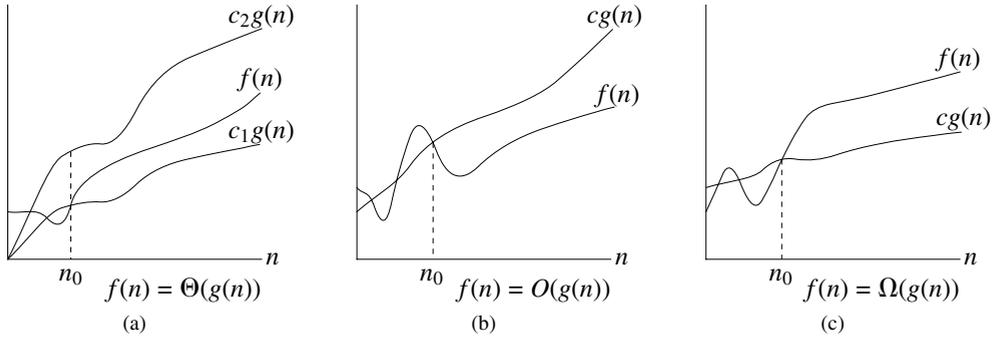


Figure 3.1 Exemples de notations Θ , O et Ω . Dans chaque partie, la valeur de n_0 est la valeur minimale possible ; n’importe quelle valeur supérieure ferait aussi l’affaire. **(a)** La notation Θ borne une fonction entre des facteurs constants. On écrit $f(n) = \Theta(g(n))$ s’il existe des constantes positives n_0 , c_1 et c_2 telles que, à droite de n_0 , la valeur de $f(n)$ soit toujours comprise entre $c_1g(n)$ et $c_2g(n)$ inclus. **(b)** La notation O donne une borne supérieure pour une fonction à un facteur constant près. On écrit $f(n) = O(g(n))$ s’il existe des constantes positives n_0 et c telles que, à droite de n_0 , la valeur de $f(n)$ soit toujours inférieure ou égale à $cg(n)$. **(c)** La notation Ω donne une borne inférieure pour une fonction à un facteur constant près. On écrit $f(n) = \Omega(g(n))$ s’il existe des constantes positives n_0 et c telles que, à droite de n_0 , la valeur de $f(n)$ soit toujours supérieure ou égale à $cg(n)$.

Au chapitre 2, nous avons défini de la manière informelle suivante la notation Θ : on élimine les termes d’ordre inférieur et on ignore le coefficient du terme d’ordre supérieur. Justifions brièvement cette définition intuitive en utilisant la définition formelle pour montrer que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Pour ce faire, on doit déterminer des constantes positives c_1 , c_2 et n_0 telles que

$$c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$$

pour tout $n \geq n_0$. En divisant par n^2 , on obtient

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

On peut s’arranger pour que le membre droit de l’inégalité soit valide pour toute valeur de $n \geq 1$, et ce en choisissant $c_2 \geq 1/2$. De même, on peut faire en sorte que le membre gauche de l’inégalité soit valide pour toute valeur de $n \geq 7$, en choisissant $c_1 \leq 1/14$. Donc, en prenant $c_1 = 1/14$, $c_2 = 1/2$ et $n_0 = 7$, on peut vérifier que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. D’autres choix sont bien entendu possibles pour les constantes ; l’important est qu’il existe au moins une possibilité. Notez que ces constantes dépendent de la fonction $\frac{1}{2}n^2 - 3n$; une autre fonction appartenant à $\Theta(n^2)$ exigerait normalement des constantes différentes.

On peut également utiliser la définition formelle pour vérifier que $6n^3 \neq \Theta(n^2)$. Supposons qu’il existe c_2 et n_0 tels que $6n^3 \leq c_2n^2$ pour tout $n \geq n_0$. On a alors $n \leq c_2/6$, ce qui ne peut pas être vrai pour n arbitrairement grand, puisque c_2 est constant.

Intuitivement, on peut ignorer les termes d'ordre inférieur d'une fonction asymptotiquement positive pour le calcul de bornes asymptotiquement approchées, car ils ne sont pas significatifs pour n grand. Une petite partie du terme d'ordre supérieur suffit à rendre négligeables les termes d'ordre inférieur. Donc, si l'on donne à c_1 une valeur légèrement plus petite que le coefficient du terme d'ordre supérieur et à c_2 une valeur légèrement plus grande, on arrive à satisfaire les inégalités sous-jacentes à la notation Θ . On peut donc ignorer le coefficient du terme d'ordre supérieur, puisque cela ne modifie c_1 et c_2 que d'un facteur constant égal à ce coefficient.

Comme exemple, considérons une fonction quadratique quelconque $f(n) = an^2 + bn + c$, où a , b et c sont des constantes, et $a > 0$. Si l'on néglige les termes d'ordre inférieur et la constante, on obtient $f(n) = \Theta(n^2)$. Formellement, pour arriver à la même conclusion, on prend $c_1 = a/4$, $c_2 = 7a/4$ et $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$. Le lecteur pourra vérifier que $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ pour tout $n \geq n_0$. Plus généralement, pour tout polynôme $p(n) = \sum_{i=0}^d a_i n^i$, où les a_i sont des constantes et $a_d > 0$, on a $p(n) = \Theta(n^d)$ (voir problème 3.1).

Comme toute constante est un polynôme de degré 0, on peut exprimer toute fonction constante par $\Theta(n^0)$ ou $\Theta(1)$. La dernière notation est légèrement abusive, car on ne sait pas quelle est la variable qui tend vers l'infini ⁽²⁾. On utilisera souvent la notation $\Theta(1)$ pour signifier indifféremment une constante ou un fonction constante d'une certaine variable.

b) Notation O

La notation Θ borne asymptotiquement une fonction à la fois par excès et par défaut. Quand on ne dispose que d'une **borne supérieure asymptotique**, on utilise la notation O . Pour une fonction $g(n)$ donnée, on note $O(g(n))$ (prononcer « grand O de g de n » ou « O de g de n ») l'ensemble de fonctions suivant :

$$O(g(n)) = \{f(n) : \text{il existe des constantes positives } c \text{ et } n_0 \text{ telles que} \\ 0 \leq f(n) \leq cg(n) \text{ pour tout } n \geq n_0\}.$$

La notation O sert à majorer une fonction, à un facteur constant près. La figure 3.1(b) montre la signification intuitive de la notation O . Pour toutes les valeurs de n situées à droite de n_0 , la valeur de la fonction $f(n)$ est inférieure ou égale à $cg(n)$.

Pour indiquer qu'une fonction $f(n)$ est membre de l'ensemble $O(g(n))$, on écrit $f(n) = O(g(n))$. On remarquera que $f(n) = \Theta(g(n))$ implique $f(n) = O(g(n))$, puisque la notation Θ est une notion plus forte que la notation O . En termes de théorie des ensembles, on a $\Theta(g(n)) \subseteq O(g(n))$. Donc, notre preuve que toute fonction quadratique $an^2 + bn + c$, avec $a > 0$, est dans $\Theta(n^2)$ prouve également que toute fonction

(2) Le véritable problème vient de ce que la notation usuelle pour les fonctions ne distingue pas les fonctions des valeurs. En λ -calcul, les paramètres d'une fonction sont clairement spécifiés : la fonction n^2 pourrait s'écrire $\lambda n.n^2$, voire $\lambda r.r^2$. Cela dit, adopter une notation plus rigoureuse compliquerait les manipulations algébriques et on tolère donc cet abus.

quadratique de ce genre est dans $O(n^2)$. Ce qui peut paraître plus surprenant, c'est que toute fonction *linéaire* $an + b$ est aussi dans $O(n^2)$, ce que l'on peut aisément vérifier en prenant $c = a + |b|$ et $n_0 = 1$.

Le lecteur qui a déjà rencontré la notation O pourrait s'étonner que nous écrivions, par exemple, $n = O(n^2)$. Dans la littérature, la notation O sert parfois de manière informelle à décrire des bornes asymptotiquement approchées, concept que nous avons, pour notre part, défini à l'aide de la notation Θ . Dans ce livre, lorsque nous écrivons $f(n) = O(g(n))$, nous disons simplement qu'un certain multiple constant de $g(n)$ est une borne supérieure de $f(n)$, sans indiquer quoi que ce soit sur le degré d'approximation de la borne supérieure. La distinction entre bornes supérieures asymptotiques et bornes asymptotiquement approchées est à présent devenue classique dans la littérature sur les algorithmes.

La notation O permet souvent de décrire le temps d'exécution d'un algorithme rien qu'en étudiant la structure globale de l'algorithme. Par exemple, la structure de boucles imbriquées de l'algorithme du tri par insertion (chapitre 2) donne immédiatement une borne supérieure en $O(n^2)$ pour le temps d'exécution du cas le plus défavorable : le coût de chaque itération de la boucle intérieure est borné supérieurement par $O(1)$ (constante), les indices i et j valent tous deux au plus n , et la boucle intérieure est exécutée au plus une fois pour chacune des n^2 paires de valeurs de i et j .

Puisque la notation O décrit une borne supérieure, quand on l'utilise pour borner le temps d'exécution du cas le plus défavorable d'un algorithme, on borne donc aussi le temps d'exécution de cet algorithme pour des entrées quelconques. Ainsi, la borne $O(n^2)$ concernant le cas le plus défavorable du tri par insertion s'applique à toutes les entrées possibles. En revanche, la borne $\Theta(n^2)$ concernant le temps d'exécution du tri par insertion dans le cas le plus défavorable n'implique pas que, pour *chaque* entrée, le tri par insertion ait un temps d'exécution borné par $\Theta(n^2)$. Par exemple, nous avons vu au chapitre 2 que, si l'entrée était déjà triée, le tri par insertion s'exécutait avec un temps en $\Theta(n)$.

En toute rigueur, c'est un abus de langage que de dire que le temps d'exécution du tri par insertion est $O(n^2)$; en effet, pour n donné, le véritable temps d'exécution dépend de la nature de l'entrée de taille n . Quand nous disons « le temps d'exécution est $O(n^2)$ », nous signifions qu'il existe une fonction $f(n)$ qui est $O(n^2)$ et telle que, pour un n fixé, pour toute entrée de taille n , le temps d'exécution est borné supérieurement par $f(n)$. Cela revient au même de dire que le temps d'exécution du cas le plus défavorable est $O(n^2)$.

c) Notation Ω

De même que la notation O fournit une borne *supérieure* asymptotique pour une fonction, la notation Ω fournit une **borne inférieure asymptotique**. Pour une fonction $g(n)$ donnée, on note $\Omega(g(n))$ (prononcer « grand oméga de g de n » ou « oméga de g

de $n \gg$) l'ensemble des fonctions suivant :

$$\Omega(g(n)) = \{f(n) : \text{il existe des constantes positives } c \text{ et } n_0 \text{ telles que} \\ 0 \leq cg(n) \leq f(n) \text{ pour tout } n \geq n_0\}.$$

La signification intuitive de la notation Ω est illustrée par la figure 3.1(c). Pour toutes les valeurs de n situées à droite de n_0 , la valeur de $f(n)$ est supérieure ou égale à $cg(n)$.

A partir des définitions des notations asymptotiques vues jusqu'ici, il est facile de prouver le théorème important suivant (voir exercice 3.1.5).

Théorème 3.1 *Pour deux fonctions quelconques $f(n)$ et $g(n)$, on a $f(n) = \Theta(g(n))$ si et seulement si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.*

Comme exemple d'application de ce théorème, notre démonstration que $an^2 + bn + c = \Theta(n^2)$ pour toutes constantes a , b et c , avec $a > 0$, implique immédiatement que $an^2 + bn + c = \Omega(n^2)$ et que $an^2 + bn + c = O(n^2)$. Dans la pratique, au lieu d'utiliser le théorème 3.1 pour obtenir des bornes supérieure et inférieure asymptotiques à partir de bornes asymptotiquement approchées, comme nous l'avons fait dans cet exemple, on l'utilise plutôt pour calculer des bornes asymptotiquement approchées à partir de bornes supérieure et inférieure asymptotiques.

Puisque la notation Ω décrit une borne inférieure, quand on l'utilise pour borner le temps d'exécution d'un algorithme dans le cas optimal, on borne alors également le temps d'exécution de l'algorithme pour des entrées arbitraires. Par exemple, le temps d'exécution du tri par insertion dans cas optimal est $\Omega(n)$, ce qui implique que le temps d'exécution du tri par insertion est $\Omega(n)$.

Le temps d'exécution du tri par insertion est donc compris entre $\Omega(n)$ et $O(n^2)$, puisqu'il se situe quelque part entre une fonction linéaire de n et une fonction quadratique de n . De plus, ces bornes sont asymptotiquement aussi approchées que possible : par exemple, le temps d'exécution du tri par insertion n'est pas $\Omega(n^2)$, car le tri prend un temps $\Theta(n)$ lorsque l'entrée est déjà triée. Cependant, il n'est pas faux de dire que le temps d'exécution du tri par Insertion *dans le cas le plus défavorable* est $\Omega(n^2)$, puisqu'il existe une entrée pour laquelle l'algorithme dure un temps $\Omega(n^2)$. Quand on dit que le *temps d'exécution* (tout court) d'un algorithme est $\Omega(g(n))$, on signifie que, pour n suffisamment grand, *quelle que soit l'entrée de taille n choisie pour une valeur quelconque de n* , le temps d'exécution pour cette entrée est au moins un multiple constant de $g(n)$.

d) Notation asymptotique dans les équations et les inégalités

Nous avons déjà vu comment les notations asymptotiques s'utilisent dans les formules mathématiques. Par exemple, quand nous avons introduit la notation O , nous avons écrit « $n = O(n^2)$ ». On aurait pu écrire également $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Quelle interprétation donner à ces formules ?

Lorsque la notation asymptotique se trouve isolée dans le membre droit d'une équation (ou d'une inégalité), comme dans $n = O(n^2)$, le signe égal est, comme nous

l'avons vu, synonyme d'appartenance : $n \in O(n^2)$. En général, toutefois, quand une notation asymptotique apparaît dans une formule, on l'interprète comme représentant une certaine fonction anonyme. Par exemple, la formule $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ signifie que $2n^2 + 3n + 1 = 2n^2 + f(n)$, où $f(n)$ est une fonction particulière de l'ensemble $\Theta(n)$. Ici $f(n) = 3n + 1$, qui appartient bien à $\Theta(n)$.

Cet emploi de la notation asymptotique aide à éliminer les détails superflus dans une équation. Par exemple, au chapitre 2, nous avons exprimé le temps d'exécution du tri par fusion dans le cas le plus défavorable sous la forme de la récurrence

$$T(n) = 2T(n/2) + \Theta(n).$$

Si l'on ne s'intéresse qu'au comportement asymptotique de $T(n)$, il n'y a aucune raison de spécifier tous les termes d'ordre inférieur ; l'on considère qu'ils sont tous compris dans la fonction anonyme représentée par le terme $\Theta(n)$.

On considère que le nombre de fonctions anonymes dans une expression est égal au nombre d'apparitions de la notation asymptotique. Par exemple, l'expression

$$\sum_{i=1}^n O(i),$$

ne contient qu'une seule fonction anonyme (une fonction de i). Cette expression n'est donc *pas* la même que $O(1) + O(2) + \dots + O(n)$, qui n'a pas d'interprétation très nette.

Dans certains cas, une notation asymptotique apparaît dans le membre gauche d'une équation, comme dans

$$2n^2 + \Theta(n) = \Theta(n^2).$$

On interprète ce type d'équation à l'aide de la règle suivante : *Quelle que soit la manière dont on choisit les fonctions anonymes à gauche du signe égal, il existe une manière de choisir les fonctions anonymes à droite du signe égal qui rende l'équation valide.* La signification de notre exemple est donc que, pour une fonction *quelconque* $f(n) \in \Theta(n)$, il existe une *certaine* fonction $g(n) \in \Theta(n^2)$ telle que $2n^2 + f(n) = g(n)$ pour tout n . Autrement dit, le membre droit d'une équation fournit un niveau de détail plus grossier que le membre gauche.

On peut enchaîner plusieurs relations de ce type, comme dans

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

On peut interpréter chaque équation séparément, grâce à la règle précédente. La première équation dit qu'il existe une *certaine* fonction $f(n) \in \Theta(n)$ telle que $2n^2 + 3n + 1 = 2n^2 + f(n)$ pour tout n . La seconde équation dit que, pour *toute* fonction $g(n) \in \Theta(n)$ (par exemple, la fonction $f(n)$ précédente), il existe une *certaine* fonction $h(n) \in \Theta(n^2)$ telle que $2n^2 + g(n) = h(n)$ pour tout n . On notera que cette interprétation implique que $2n^2 + 3n + 1 = \Theta(n^2)$, ce que d'ailleurs l'enchaînement des équations suggère intuitivement.

e) *Notation o*

La borne supérieure asymptotique fournie par la notation O peut être ou non asymptotiquement approchée. La borne $2n^2 = O(n^2)$ est asymptotiquement approchée, mais la borne $2n = O(n^2)$ ne l'est pas. On utilise la notation o pour noter une borne supérieure qui n'est pas asymptotiquement approchée. On définit formellement $o(g(n))$ (« petit o de g de n ») comme étant l'ensemble

$$o(g(n)) = \{f(n) : \text{pour toute constante } c > 0, \text{ il existe une constante } n_0 > 0 \text{ telle que } 0 \leq f(n) < cg(n) \text{ pour tout } n \geq n_0\} .$$

Par exemple, $2n = o(n^2)$ mais $2n^2 \neq o(n^2)$.

Les définitions des notations O et o se ressemblent. La différence principale est que dans $f(n) = O(g(n))$, la borne $0 \leq f(n) \leq cg(n)$ est valable pour une *certaine* constante $c > 0$, alors que dans $f(n) = o(g(n))$, la borne $0 \leq f(n) < cg(n)$ est valable pour *toutes* les constantes $c > 0$. De manière intuitive, avec la notation o , la fonction $f(n)$ devient négligeable par rapport à $g(n)$ quand n tend vers l'infini ; en d'autres termes

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 . \quad (3.1)$$

Certains auteurs utilisent cette limite comme définition de la notation o ; la définition donnée dans ce livre impose en outre que les fonctions anonymes soient asymptotiquement positives.

f) *Notation ω*

Par analogie, la notation ω est à la notation Ω ce que la notation o est à la notation O . On utilise la notation ω pour indiquer une borne inférieure qui n'est pas asymptotiquement approchée. Une façon de la définir est

$$f(n) \in \omega(g(n)) \text{ si et seulement si } g(n) \in o(f(n)) .$$

Formellement, on définit $\omega(g(n))$ (« petit oméga de g de n ») comme l'ensemble

$$\omega(g(n)) = \{f(n) : \text{pour toute constante } c > 0, \text{ il existe une constante } n_0 > 0 \text{ telle que } 0 \leq cg(n) < f(n) \text{ pour tout } n \geq n_0\} .$$

Par exemple, $n^2/2 = \omega(n)$ mais $n^2/2 \neq \omega(n^2)$. La relation $f(n) = \omega(g(n))$ implique que

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty ,$$

si la limite existe. C'est-à-dire que $f(n)$ devient arbitrairement grande par rapport à $g(n)$ quand n tend vers l'infini.

g) Comparaison de fonctions

Nombre de propriétés relationnelles des nombres réels s'appliquent aussi aux comparaisons asymptotiques. Dans ce qui suit, on supposera que $f(n)$ et $g(n)$ sont asymptotiquement positives.

Transitivité :

$$\begin{aligned} f(n) = \Theta(g(n)) \text{ et } g(n) = \Theta(h(n)) & \text{ implique } f(n) = \Theta(h(n)) , \\ f(n) = O(g(n)) \text{ et } g(n) = O(h(n)) & \text{ implique } f(n) = O(h(n)) , \\ f(n) = \Omega(g(n)) \text{ et } g(n) = \Omega(h(n)) & \text{ implique } f(n) = \Omega(h(n)) , \\ f(n) = o(g(n)) \text{ et } g(n) = o(h(n)) & \text{ implique } f(n) = o(h(n)) , \\ f(n) = \omega(g(n)) \text{ et } g(n) = \omega(h(n)) & \text{ implique } f(n) = \omega(h(n)) . \end{aligned}$$

Réflexivité :

$$\begin{aligned} f(n) &= \Theta(f(n)) , \\ f(n) &= O(f(n)) , \\ f(n) &= \Omega(f(n)) . \end{aligned}$$

Symétrie :

$$f(n) = \Theta(g(n)) \text{ si et seulement si } g(n) = \Theta(f(n)) .$$

Symétrie transposée :

$$\begin{aligned} f(n) = O(g(n)) \text{ si et seulement si } g(n) = \Omega(f(n)) , \\ f(n) = o(g(n)) \text{ si et seulement si } g(n) = \omega(f(n)) . \end{aligned}$$

Comme ces propriétés sont vraies pour des notations asymptotiques, on peut dégager une analogie entre la comparaison asymptotique de deux fonctions f et g et la comparaison de deux réels a et b :

$$\begin{aligned} f(n) = O(g(n)) & \approx a \leq b , \\ f(n) = \Omega(g(n)) & \approx a \geq b , \\ f(n) = \Theta(g(n)) & \approx a = b , \\ f(n) = o(g(n)) & \approx a < b , \\ f(n) = \omega(g(n)) & \approx a > b . \end{aligned}$$

Nous dirons que $f(n)$ est *asymptotiquement inférieure* à $g(n)$ si $f(n) = o(g(n))$, et que $f(n)$ est *asymptotiquement supérieure* à $g(n)$ si $f(n) = \omega(g(n))$. Toutefois, il existe une propriété des nombres réels qui n'est pas applicable à la notation asymptotique :

Trichotomie : Pour deux nombres réels quelconques a et b , une et une seule des propositions suivantes est vraie : $a < b$, $a = b$ ou $a > b$.

Alors que deux nombres réels peuvent toujours être comparés, deux fonctions ne sont pas toujours comparables asymptotiquement. En d'autres termes, pour deux fonctions $f(n)$ et $g(n)$, il se peut que l'on n'ait ni $f(n) = O(g(n))$, ni $f(n) = \Omega(g(n))$. Par

exemple, les fonctions n et $n^{1+\sin n}$ ne peuvent pas être comparées à l'aide d'une notation asymptotique, puisque la valeur de l'exposant dans $n^{1+\sin n}$ oscille entre 0 et 2 en prenant successivement toutes les valeurs de cet intervalle.

Exercices

3.1.1 Soient $f(n)$ et $g(n)$ des fonctions asymptotiquement non négatives. En s'aidant de la définition de base de la notation Θ , prouver que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1.2 Montrer que, pour deux constantes réelles a et b quelconques avec $b > 0$, l'on a

$$(n + a)^b = \Theta(n^b). \quad (3.2)$$

3.1.3 Expliquer pourquoi l'affirmation « Le temps d'exécution de l'algorithme A est au moins $O(n^2)$ » n'a pas de sens.

3.1.4 Est-ce que $2^{n+1} = O(2^n)$? Est-ce que $2^{2n} = O(2^n)$?

3.1.5 Démontrer le théorème 3.1.

3.1.6 Démontrer que le temps d'exécution d'un algorithme est $\Theta(g(n))$ si et seulement si son temps d'exécution dans le cas le plus défavorable est $O(g(n))$ et son temps d'exécution dans le meilleur des cas est $\Omega(g(n))$.

3.1.7 Démontrer que $o(g(n)) \cap \omega(g(n))$ est l'ensemble vide.

3.1.8 On peut étendre notre notation au cas de deux paramètres n et m qui tendent vers l'infini indépendamment à des vitesses différentes. Pour une fonction donnée $g(n, m)$, on note $O(g(n, m))$ l'ensemble des fonctions

$$O(g(n, m)) = \{f(n, m) : \text{il existe des constantes positives } c, n_0 \text{ et } m_0 \text{ telles que } 0 \leq f(n, m) \leq cg(n, m) \text{ pour tout } n \geq n_0 \text{ et tout } m \geq m_0\}.$$

Donner des définitions correspondantes pour $\Omega(g(n, m))$ et $\Theta(g(n, m))$.

3.2 NOTATIONS STANDARD ET FONCTIONS CLASSIQUES

Cette section va revoir certaines fonctions et notations mathématiques standard et examiner les relations entre elles. Elle va également illustrer l'utilisation des notations asymptotiques.

a) *Monotonie*

Une fonction $f(n)$ est **monotone croissante** si $m \leq n$ implique $f(m) \leq f(n)$. De même, elle est **monotone décroissante** si $m \leq n$ implique $f(m) \geq f(n)$. Une fonction $f(n)$ est **strictement croissante** si $m < n$ implique $f(m) < f(n)$ et **strictement décroissante** si $m < n$ implique $f(m) > f(n)$.

b) *Parties entières*

Pour tout réel x , on représente le plus grand entier inférieur ou égal à x par $\lfloor x \rfloor$ (lire « partie entière de x ») et le plus petit entier supérieur ou égal à x par $\lceil x \rceil$ (lire « partie entière supérieure de x »). Pour tout x réel, on a

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 . \quad (3.3)$$

Pour un entier n quelconque, on a

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n ,$$

et pour un réel $n \geq 0$ et des entiers $a, b > 0$, l'on a

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil , \quad (3.4)$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor , \quad (3.5)$$

$$\lceil a/b \rceil \leq (a + (b - 1))/b , \quad (3.6)$$

$$\lfloor a/b \rfloor \geq ((a - (b - 1))/b) . \quad (3.7)$$

Les fonctions $\lfloor x \rfloor$ et $\lceil x \rceil$ sont toutes les deux monotones croissantes.

c) *Congruences*

Pour tout entier a et pour tout entier positif n , la valeur $a \bmod n$ est le **reste** du quotient a/n :

$$a \bmod n = a - \lfloor a/n \rfloor n . \quad (3.8)$$

Partant de la notion connue de reste de la division d'un entier par un autre, il est commode de fournir une notation spéciale pour indiquer l'égalité des restes. Si $(a \bmod n) = (b \bmod n)$, on écrit $a \equiv b \pmod{n}$ et l'on dit que a est **congru** à b modulo n . En d'autres termes, $a \equiv b \pmod{n}$ si a et b ont le même reste dans la division par n . Il revient au même de dire que $a \equiv b \pmod{n}$ si et seulement si n est un diviseur de $b - a$. On écrira $a \not\equiv b \pmod{n}$ si a n'est pas congru à b modulo n .

d) *Polynômes*

Étant donné un entier non négatif d , un **polynôme en n de degré d** est une fonction $p(n)$ de la forme

$$p(n) = \sum_{i=0}^d a_i n^i ,$$

les constantes a_0, a_1, \dots, a_d étant les *coefficients* du polynôme et $a_d \neq 0$. Un polynôme est *asymptotiquement positif* si et seulement si $a_d > 0$. Pour un polynôme asymptotiquement positif $p(n)$ de degré d , on a $p(n) = \Theta(n^d)$. Pour toute constante réelle $a \geq 0$, la fonction n^a est monotone croissante ; pour toute constante réelle $a \leq 0$, la fonction n^a est monotone décroissante. On dit qu'une fonction $f(n)$ a une *borne polynomiale* si $f(n) = O(n^k)$ pour une certaine constante k .

e) Exponentielles

Pour tous réels $a \neq 0, m$ et n , on a les identités suivantes :

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

Pour tout n et tout $a \geq 1$, la fonction a^n est monotone croissante en n . Quand cela s'avérera pratique, nous supposons que $0^0 = 1$.

Il existe une relation entre les vitesses de croissance des polynômes et des exponentielles. Pour toutes constantes réelles a et b telles que $a > 1$, on a :

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0, \quad (3.9)$$

d'où l'on peut conclure que

$$n^b = o(a^n).$$

Ainsi, toute fonction exponentielle dont la base est strictement plus grande que 1 croît plus vite que toute fonction polynomiale.

En notant e le réel $2,71828\dots$, base de la fonction logarithme népérien (naturel), on a pour tout réel x :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{i=0}^{\infty} \frac{x^i}{i!}, \quad (3.10)$$

où « ! » représente la fonction factorielle définie plus loin dans cette section. Pour tout réel x , on a l'inégalité

$$e^x \geq 1 + x, \quad (3.11)$$

(l'égalité n'a lieu que pour $x = 0$). Lorsque $|x| \leq 1$, on a l'approximation

$$1 + x \leq e^x \leq 1 + x + x^2. \quad (3.12)$$

Quand $x \rightarrow 0$, l'approximation de e^x par $1 + x$ est assez bonne :

$$e^x = 1 + x + \Theta(x^2).$$

(Dans cette équation, la notation asymptotique sert à décrire le comportement aux limites quand $x \rightarrow 0$ et non quand $x \rightarrow \infty$.) On a pour tout x :

$$\lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n = e^x. \quad (3.13)$$

f) Logarithmes

On utilisera les notations suivantes :

$$\lg n = \log_2 n \quad (\text{logarithme de base 2}) ,$$

$$\ln n = \log_e n \quad (\text{logarithme népérien}) ,$$

$$\lg^k n = (\lg n)^k \quad (\text{exponentiation}) ,$$

$$\lg \lg n = \lg(\lg n) \quad (\text{composition}) .$$

Nous adopterons la convention notationnelle importante selon laquelle *les fonctions logarithme s'appliquent uniquement au terme qui suit dans la formule*, de sorte que $\lg n + k$ signifiera $(\lg n) + k$ et non $\lg(n + k)$. Si nous prenons $b > 1$ constant, alors pour $n > 0$ la fonction $\log_b n$ est strictement croissante. Pour tous réels $a > 0$, $b > 0$, $c > 0$ et n , on a :

$$\begin{aligned} a &= b^{\log_b a} , \\ \log_c(ab) &= \log_c a + \log_c b , \\ \log_b a^n &= n \log_b a , \\ \log_b a &= \frac{\log_c a}{\log_c b} , \end{aligned} \quad (3.14)$$

$$\begin{aligned} \log_b(1/a) &= -\log_b a , \\ \log_b a &= \frac{1}{\log_a b} , \\ a^{\log_b c} &= c^{\log_b a} , \end{aligned} \quad (3.15)$$

avec, dans chacune de ces équations, une base de logarithme qui n'est pas 1.

D'après l'équation (3.14), changer la base d'un logarithme ne modifie la valeur du logarithme que d'un facteur constant ; nous utiliserons donc souvent la notation « $\lg n$ » lorsque nous ne nous soucierons pas des facteurs constants, par exemple dans la notation O . Pour les informaticiens, 2 est la base la plus naturelle pour les logarithmes ; en effet, nombreux sont les algorithmes et structures de données qui impliquent la séparation d'un problème en deux parties.

Il existe un développement en série simple pour $\ln(1+x)$ quand $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

On a également les inégalités suivantes pour $x > -1$:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x, \quad (3.16)$$

(l'égalité n'a lieu que pour $x = 0$).

On dit qu'une fonction $f(n)$ a une **borne polylogarithmique** si $f(n) = O(\lg^k n)$ pour une certaine constante k . On peut mettre en relation la croissance des polynômes et des polylogarithmes en substituant $\lg n$ à n et 2^a à a dans l'équation (3.9), ce qui donne

$$\lim_{n \rightarrow \infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n \rightarrow \infty} \frac{\lg^b n}{n^a} = 0.$$

De cette limite, on peut conclure que

$$\lg^b n = o(n^a)$$

pour toute constante $a > 0$. Ainsi, toute fonction polynomiale positive croît plus vite que toute fonction polylogarithmique.

g) Factorielles

La notation $n!$ (lire « factorielle n ») est définie comme suit pour les entiers $n \geq 0$:

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot (n-1)! & \text{si } n > 0. \end{cases}$$

Donc, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Une borne supérieure faible de la fonction factorielle est $n! \leq n^n$, puisque chacun des n termes du produit factoriel vaut au plus n . La **formule de Stirling**,

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right), \quad (3.17)$$

e étant la base du logarithme naturel, donne une borne supérieure plus serrée ainsi qu'une borne inférieure. On peut montrer (voir exercice 3.2.3)

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n), \end{aligned} \quad (3.18)$$

la formule de Stirling servant ici à prouver l'équation (3.18). L'équation suivante est également vraie pour tout $n \geq 1$:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha_n} \quad (3.19)$$

avec

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}. \quad (3.20)$$

h) *Itération fonctionnelle*

La notation $f^{(i)}(n)$ représente la fonction $f(n)$ appliquée, de manière itérative, i fois à une valeur initiale n . De manière plus formelle, soit $f(n)$ une fonction définie sur les réels. Pour les entiers non négatifs i , on définit récursivement

$$f^{(i)}(n) = \begin{cases} n & \text{si } i = 0, \\ f(f^{(i-1)}(n)) & \text{si } i > 0. \end{cases}$$

Par exemple, si $f(n) = 2n$ alors $f^{(i)}(n) = 2^i n$.

i) *Fonction logarithme itéré*

La notation $\lg^* n$ (lire « log étoile de n ») représente le logarithme itéré, ainsi défini : soit $\lg^{(i)} n$ la fonction définie itérativement comme précédemment exposé, où ici $f(n) = \lg n$. Comme le logarithme n'est défini que pour les nombres positifs, $\lg^{(i)} n$ n'est définie que si $\lg^{(i-1)} n > 0$. Ne confondez pas $\lg^{(i)} n$ (fonction logarithme appliquée i fois successivement, l'argument de départ étant n) et $\lg^i n$ (logarithme de n élevé à la puissance i th). La fonction logarithme itéré est définie ainsi :

$$\lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\} .$$

Le logarithme itéré est une fonction à croissance *très* lente :

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

Le nombre d'atomes de l'univers observable étant estimé à environ 10^{80} , valeur très inférieure à 2^{65536} , il est rare de tomber sur une entrée de taille n tel que $\lg^* n > 5$.

j) *Nombres de Fibonacci*

Les **nombres de Fibonacci** sont définis par la récurrence suivante :

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \quad \text{pour } i \geq 2. \end{aligned} \tag{3.21}$$

Chaque nombre de Fibonacci est donc la somme des deux précédents, ce qui donne la suite

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Les nombres Fibonacci sont étroitement liés au **nombre d'or** ϕ et à son conjugué $\hat{\phi}$, donnés par les formules suivantes :

$$\begin{aligned}\phi &= \frac{1 + \sqrt{5}}{2} & (3.22) \\ &= 1.61803 \dots, \\ \hat{\phi} &= \frac{1 - \sqrt{5}}{2} \\ &= -.61803 \dots.\end{aligned}$$

Plus précisément, on a

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}, \quad (3.23)$$

ce qu'on peut prouver par récurrence (exercice 3.2.6). Comme $|\hat{\phi}| < 1$, on a $|\hat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$; de sorte que le i -ème nombre de Fibonacci F_i est égal à $\phi^i/\sqrt{5}$, arrondi à l'entier le plus proche. Les nombres de Fibonacci croissent donc de façon exponentielle.

Exercices

3.2.1 Montrer que, si $f(n)$ et $g(n)$ sont des fonctions monotones croissantes, alors les fonctions $f(n) + g(n)$ et $f(g(n))$ le sont également; montrer que, si $f(n)$ et $g(n)$ sont en outre non négatives, $f(n) \cdot g(n)$ est monotone croissante.

3.2.2 Démontrer l'équation (3.15).

3.2.3 Prouver l'équation (3.18). Montrer aussi que $n! = \omega(2^n)$ et $n! = o(n^n)$.

3.2.4 * La fonction $\lceil \lg n \rceil!$ a-t-elle une borne polynomiale? Et la fonction $\lceil \lg \lg n \rceil!$?

3.2.5 * Laquelle de ces deux fonctions est la plus grande asymptotiquement: $\lg(\lg^* n)$ ou $\lg^*(\lg n)$?

3.2.6 Démontrer par récurrence que le i -ème nombre de Fibonacci satisfait à l'égalité

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}},$$

où ϕ est le nombre d'or et $\hat{\phi}$ son conjugué.

3.2.7 Démontrer que, pour $i \geq 0$, le $(i + 2)$ -ème nombre de Fibonacci satisfait à l'inégalité $F_{i+2} \geq \phi^i$.

PROBLÈMES

3.1. Comportement asymptotique des polynômes

Soit

$$p(n) = \sum_{i=0}^d a_i n^i,$$

avec $a_d > 0$, un polynôme en n de degré d , et soit k une constante. Utiliser les définitions des notations asymptotiques pour démontrer les propriétés suivantes.

- a. Si $k \geq d$, alors $p(n) = O(n^k)$.
- b. Si $k \leq d$, alors $p(n) = \Omega(n^k)$.
- c. Si $k = d$, alors $p(n) = \Theta(n^k)$.
- d. Si $k > d$, alors $p(n) = o(n^k)$.
- e. Si $k < d$, alors $p(n) = \omega(n^k)$.

3.2. Croissances asymptotiques relatives

Indiquer, pour chaque paire d'expressions (A, B) du tableau ci-après, si A est O , o , Ω , ω ou Θ de B . On suppose que $k \geq 1$, $\varepsilon > 0$ et $c > 1$ sont des constantes. Répondre en écrivant « oui » ou « non » dans chaque case du tableau.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ε					
b.	n^k	c^n					
c.	\sqrt{n}	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg m}$	$m^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3.3. Classement par vitesses de croissance asymptotiques

- a. Ranger les fonctions suivantes par ordre de croissance ; c'est-à-dire, trouver un ordre g_1, g_2, \dots, g_{30} pour ces fonctions tel que $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, \dots , $g_{29} = \Omega(g_{30})$. Partitionner la liste en classes d'équivalence telles que $f(n)$ et $g(n)$ sont dans la même classe si et seulement si $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

- b.** Trouver une fonction non négative $f(n)$ telle que, pour toutes les fonctions $g_i(n)$ de la partie (a), $f(n)$ ne soit ni $O(g_i(n))$ ni $\Omega(g_i(n))$.

3.4. Propriétés de la notation asymptotique

Soient $f(n)$ et $g(n)$ des fonctions asymptotiquement positives. Prouver ou démentir chacune des affirmations suivantes.

- a.** $f(n) = O(g(n))$ implique $g(n) = O(f(n))$.
b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.
c. $f(n) = O(g(n))$ implique $\lg(f(n)) = O(\lg(g(n)))$, où $\lg(g(n)) \geq 1$ et $f(n) \geq 1$ pour tout n suffisamment grand.
d. $f(n) = O(g(n))$ implique $2^{f(n)} = O(2^{g(n)})$.
e. $f(n) = O((f(n))^2)$.
f. $f(n) = O(g(n))$ implique $g(n) = \Omega(f(n))$.
g. $f(n) = \Theta(f(n/2))$.
h. $f(n) + o(f(n)) = \Theta(f(n))$.

3.5. Variations sur O et Ω

Certains auteurs définissent Ω d'une façon légèrement différente de la nôtre ; notons $\overset{\infty}{\Omega}$ (lire « oméga infini ») cette autre définition. On dit que $f(n) = \overset{\infty}{\Omega}(g(n))$ s'il existe une constante positive c telle que $f(n) \geq cg(n) \geq 0$ pour un nombre infiniment grand d'entiers n .

- a.** Montrer que, pour deux fonctions quelconques $f(n)$ et $g(n)$ qui sont asymptotiquement non négatives, on a soit $f(n) = O(g(n))$, soit $f(n) = \overset{\infty}{\Omega}(g(n))$, soit les deux, alors que cela est faux si l'on utilise Ω au lieu de $\overset{\infty}{\Omega}$.
b. Expliquer quels sont les avantages et inconvénients potentiels qu'entraîne l'utilisation de $\overset{\infty}{\Omega}$ à la place de Ω pour caractériser le temps d'exécution des programmes.

Certains auteurs définissent également O d'une façon légèrement différente de la nôtre ; notons O' cette autre définition. On dit que $f(n) = O'(g(n))$ si et seulement si $|f(n)| = O(g(n))$.

- c. Qu'advient-il, pour chaque sens du « si et seulement si » du théorème 3.1, si l'on substitue O' à O tout en continuant d'utiliser Ω ?

Certains auteurs définissent \tilde{O} (lire « O tilde ») pour signifier O sans les facteurs logarithmiques :

$$\tilde{O}(g(n)) = \{f(n) : \text{il existe des constantes positives } c, k \text{ et } n_0 \text{ telles que } 0 \leq f(n) \leq cg(n) \lg^k(n) \text{ pour tout } n \geq n_0\}.$$

- d. Définir $\tilde{\Omega}$ et $\tilde{\Theta}$ d'une manière similaire. Prouver le théorème qui est l'homologue du théorème 3.1.

3.6. Fonctions itérées

L'opérateur d'itération $*$ utilisé dans la fonction \lg^* peut s'appliquer à toute fonction monotone croissante à variable réelle. Pour une constante donnée $c \in \mathbf{R}$, on définit la fonction itérée f_c^* par

$$f_c^*(n) = \min \{i \geq 0 : f^{(i)}(n) \leq c\},$$

qui n'est pas obligée d'être bien définie dans tous les cas. En d'autres termes, la quantité $f_c^*(n)$ est le nombre de fois qu'il faut appliquer, de manière itérée, la fonction f pour faire tomber son argument à une valeur inférieure ou égale à c . Pour chacune des fonctions $f(n)$ et des constantes c suivantes, donner une borne aussi approchée que possible pour $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$n - 1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	\sqrt{n}	2	
f.	\sqrt{n}	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

NOTES

Knuth [182] fait remonter les origines de la notation O à un texte sur la théorie des nombres écrit par P. Bachmann en 1892. La notation o fut inventée par E. Landau en 1909 pour son étude sur la distribution des nombres premiers. L'emploi des notations Ω et Θ a été recommandé par Knuth [186] qui dénonce la pratique, très répandue mais peu rigoureuse, qui consiste à utiliser la notation O aussi bien pour les bornes supérieures que les bornes inférieures. Maintes personnes continuent à utiliser la notation O là où la notation Θ serait

plus appropriée. On trouvera une étude plus précise de l'histoire et du développement des notations asymptotiques dans Knuth [182, 186], ainsi que dans Brassard et Bratley [46].

Tous les auteurs ne définissent pas les notations asymptotiques de la même façon, bien que les différentes définitions se rejoignent dans la plupart des situations concrètes. Certaines de ces autres définitions englobent les fonctions qui ne sont pas asymptotiquement non négatives, à condition que leurs valeurs absolues soient bornées comme il faut.

L'équation (3.19) est due à Robbins [260]. On trouvera d'autres propriétés des fonctions mathématiques élémentaires dans tout bon manuel de mathématiques générales, dont Abramowitz et Stegun [1] ou Zwillinger [320], ou encore dans un manuel de calcul numérique, tel Apostol [18] ou Thomas et Finney [296]. Knuth [182] et Graham, Knuth et Patashnik [132] renferment quantité de choses sur les mathématiques discrètes appliquées à l'informatique.

Chapitre 4

Récurrances

Comme nous l'avons vu à la section 2.3.2, quand un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une récurrence. Une **récurrence** est une équation ou inégalité qui décrit une fonction à partir de sa valeur sur des entrées plus petites. Par exemple, nous avons vu à la section 2.3.2 que le temps $T(n)$ d'exécution du cas le plus défavorable de la procédure TRI-FUSION pouvait être décrit par la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(n/2) + \Theta(n) & \text{si } n > 1, \end{cases} \quad (4.1)$$

dont nous avons affirmé que la solution était $T(n) = \Theta(n \lg n)$.

Ce chapitre va présenter trois méthodes pour résoudre les récurrences, c'est-à-dire pour obtenir des bornes asymptotiques « Θ » ou « O » pour la solution. Dans la **méthode de substitution**, on devine une borne puis on utilise une récurrence mathématique pour démontrer la validité de la conjecture. La **méthode de l'arbre récursif** convertit la récurrence en un arbre dont les nœuds représentent les coûts induits à différents niveaux de la récursivité ; nous emploierons des techniques de bornage de sommation pour résoudre la récurrence. La **méthode générale** fournit des bornes pour les récurrences de la forme

$$T(n) = aT(n/b) + f(n),$$

où $a \geq 1$, $b > 1$ et $f(n)$ est une fonction donnée ; cette méthode oblige à traiter trois cas distincts, mais facilite la détermination des bornes asymptotiques pour nombre de récurrences simples.

a) Considérations techniques

En pratique, on néglige certains détails techniques pour énoncer et résoudre les récurrences. Par exemple, on passe souvent sur le fait que les arguments des fonctions sont des entiers. Normalement, le temps d'exécution $T(n)$ d'un algorithme n'est défini que pour n entier puisque, dans la plupart des algorithmes, la taille de l'entrée a toujours une valeur entière. Par exemple, la récurrence décrivant le temps d'exécution de TRI-FUSION dans le cas le plus défavorable devrait en fait s'écrire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{si } n > 1. \end{cases} \quad (4.2)$$

Autre classe de détails le plus souvent ignorée : les conditions aux limites. Puisque le temps d'exécution d'un algorithme sur une entrée de taille constante est une constante, les récurrences sous-jacentes au calcul du temps d'exécution des algorithmes ont généralement $T(n) = \Theta(1)$ pour n suffisamment petit. Du coup, pour simplifier, on omet généralement de définir les conditions aux limites des récurrences et l'on suppose que $T(n)$ est constant pour n petit. Par exemple, la récurrence (4.1) est généralement définie de la façon suivante :

$$T(n) = 2T(n/2) + \Theta(n), \quad (4.3)$$

sans que l'on donne explicitement des valeurs pour n petit. La raison en est la suivante : bien que changer la valeur de $T(1)$ ait pour effet de changer la solution de la récurrence, en général cette solution ne change pas de plus d'un facteur constant, ce qui ne modifie pas l'ordre de grandeur.

Quand on définit et résout des récurrences, on omet souvent les parties entières et les conditions aux limites. On va de l'avant sans se préoccuper de ces détails puis, ultérieurement, on voit s'ils sont importants ou non. En général ils ne sont pas importants, mais il est vital de savoir quand ils le sont. L'expérience aide, de même que certains théorèmes énonçant que ces détails n'affectent pas les bornes asymptotiques de moult récurrences rencontrées dans l'analyse des algorithmes (voir théorème 4.1). Dans ce chapitre, toutefois, on abordera certains de ces détails afin d'illustrer les subtilités des méthodes de résolution de récurrence.

4.1 MÉTHODE DE SUBSTITUTION

La méthode de substitution recouvre deux phases :

- 1) Conjecturer la forme de la solution.
- 2) Employer une récurrence mathématique pour trouver les constantes et prouver que la solution est correcte.

Le nom vient du fait que l'on substitue la réponse supposée à la fonction quand on applique l'hypothèse de récurrence aux valeurs plus petites. Cette méthode est puissante, mais ne peut manifestement s'utiliser que lorsque la forme de la réponse

est facile à deviner. La méthode de substitution peut servir à borner une récurrence par excès ou par défaut. Par exemple, calculons une borne supérieure pour la récurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, \quad (4.4)$$

qui ressemble aux récurrences (4.2) et (4.3). On conjecture que la solution est $T(n) = O(n \lg n)$. La méthode consiste à démontrer que $T(n) \leq cn \lg n$ pour un choix approprié de la constante $c > 0$. On commence par supposer que cette borne est valable pour $\lfloor n/2 \rfloor$, autrement dit que $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. La substitution donne

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n, \end{aligned}$$

la dernière étape étant vraie si $c \geq 1$.

L'induction mathématique exige que nous montrions que notre solution est valable pour les conditions aux limites. En général, on le fait en montrant que les conditions aux limites peuvent servir de cas initiaux pour la démonstration par récurrence. Pour la récurrence (4.4), on doit montrer qu'il est possible de choisir une constante c suffisamment grande pour que la borne $T(n) \leq cn \lg n$ soit vraie aussi pour les conditions aux limites. Cette contrainte peut parfois induire des problèmes. Supposons, pour les besoins de la discussion, que $T(1) = 1$ soit l'unique condition aux limites pour la récurrence. Alors, pour $n = 1$, la borne $T(n) \leq cn \lg n$ donne $T(1) \leq c1 \lg 1 = 0$, ce qui contredit $T(1) = 1$. Par conséquent, le cas initial de notre démonstration inductive n'est pas vérifié.

Il est facile de passer outre cette difficulté qu'il y a à prouver une hypothèse de récurrence pour une certaine condition aux limites. Par exemple, dans la récurrence (4.4), on profite de ce que la notation asymptotique nous oblige uniquement à démontrer que $T(n) \leq cn \lg n$ pour $n \geq n_0$, où n_0 est une constante de notre choix. L'idée est de supprimer, dans la démonstration par récurrence, l'obligation de considérer la délicate condition aux limites $T(1) = 1$. Observez que, pour $n > 3$, la récurrence ne dépend pas directement de $T(1)$. On peut donc remplacer $T(1)$ par $T(2)$ et $T(3)$ comme cas initiaux de la démonstration par récurrence, en faisant $n_0 = 2$. Notez que nous faisons une distinction entre le cas initial de la récurrence ($n = 1$) et les cas initiaux de la démonstration ($n = 2$ et $n = 3$). Nous déduisons de la récurrence que $T(2) = 4$ et $T(3) = 5$. Nous pouvons maintenant compléter la démonstration par récurrence que $T(n) \leq cn \lg n$ pour une certaine constante $c \geq 1$ en choisissant c suffisamment grand pour que $T(2) \leq c2 \lg 2$ et $T(3) \leq c3 \lg 3$. En fait, on constate que n'importe quel choix de $c \geq 2$ valide les cas initiaux de $n = 2$ et $n = 3$. Pour la plupart des récurrences que nous étudierons, il sera immédiat d'étendre les conditions aux limites de façon que l'hypothèse de récurrence soit vraie pour n petit.

b) De l'art de bien conjecturer

Malheureusement, il n'existe de pas de règle générale pour conjecturer les solutions des récurrences. Deviner une solution ressort de l'expérience et, parfois, de l'intuition pure. Cela dit, il existe certaines heuristiques qui vous aideront à devenir un bon devin. Vous pouvez aussi faire appel aux arbres de récursivité, que nous verrons à la section 4.2, pour générer de bonnes conjectures.

Si une récurrence ressemble à une autre rencontrée précédemment, il est raisonnable de conjecturer une solution similaire. Par exemple, la récurrence

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n ,$$

semble difficile à cause du « 17 » ajouté à l'argument de T dans le membre droit. Toutefois, on sent bien que ce terme supplémentaire n'affecte pas sensiblement la solution de la récurrence. Quand n devient grand, la différence entre $T(\lfloor n/2 \rfloor)$ et $T(\lfloor n/2 \rfloor + 17)$ n'est plus aussi importante : ces deux expressions divisent n par deux. On suppose donc que $T(n) = O(n \lg n)$, puis on vérifiera la validité de la conjecture en utilisant la méthode de substitution (voir exercice 4.1.5).

Un autre moyen de bien conjecturer consiste à trouver des bornes supérieure et inférieure très larges pour la récurrence, puis à réduire l'intervalle d'incertitude. Ainsi, on peut partir d'une borne inférieure $T(n) = \Omega(n)$ pour la récurrence (4.4) (car on a le terme n dans la récurrence), puis montrer l'existence d'une borne supérieure initiale de $T(n) = O(n^2)$. Ensuite, on peut progressivement diminuer la borne supérieure et augmenter la borne inférieure, pour finalement converger vers la bonne solution asymptotiquement approchée qui est $T(n) = \Theta(n \lg n)$.

c) Subtilités

Il peut advenir que l'on arrive à conjecturer la borne asymptotique de la solution d'une récurrence mais que l'on n'arrive pas, d'une façon ou d'une autre, à ajuster les différentes valeurs au niveau de la récurrence. En général, le problème vient de ce que l'hypothèse de récurrence n'est pas assez forte pour que l'on puisse valider la borne précise. Quand on rencontre ce type d'obstacle, il suffit souvent de revoir l'hypothèse, en y incluant la soustraction d'un terme d'ordre inférieur, pour que les paramètres s'ajustent correctement.

Considérons la récurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 .$$

On conjecture que la solution est $O(n)$ et l'on essaye de montrer que $T(n) \leq cn$ pour un choix approprié de la constante c . Quand on substitue la conjecture dans la récurrence, on obtient

$$\begin{aligned} T(n) &\leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

Ce qui n'implique pas que $T(n) \leq cn$ pour n'importe quel c . Il est tentant d'essayer une expression d'ordre supérieur, par exemple $T(n) = O(n^2)$, qu'on choisit ad hoc. Mais en fait, notre conjecture $T(n) = O(n)$ est bonne. Toutefois, pour le démontrer, il faut faire une hypothèse de récurrence plus forte.

Intuitivement, notre conjecture est presque bonne : on ne diverge que de la constante 1, qui est un terme d'ordre inférieur. Néanmoins, la récurrence mathématique ne marchera que si nous prouvons la forme exacte de l'hypothèse de récurrence. On contourne la difficulté en retouchant un terme d'ordre inférieur à notre précédente hypothèse. Celle-ci devient $T(n) \leq cn - b$, où $b \geq 0$ est constant. On a maintenant :

$$\begin{aligned} T(n) &\leq (c \lfloor n/2 \rfloor - b) + (c \lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b, \end{aligned}$$

tant que $b \geq 1$. Comme précédemment, la constante c doit être choisie suffisamment grande pour que soient respectées les conditions aux limites.

La plupart des gens trouvent que l'idée de soustraire un terme d'ordre inférieur va contre l'intuition. Mais, si le mécanisme mathématique ne fonctionne pas, pourquoi est-ce que l'on ne renforcerait pas la conjecture ? La démarche repose sur le fait même que l'on est en train d'utiliser la récurrence mathématique : pour prouver quelque chose de plus fort pour une valeur donnée, on suppose quelque chose de plus fort pour les valeurs plus petites.

d) Pièges

Il est facile de se fourvoyer dans l'emploi de la notation asymptotique. Par exemple, dans la récurrence (4.4), on peut « démontrer » à tort que $T(n) = O(n)$ en conjecturant $T(n) \leq cn$ et en raisonnant de la façon suivante :

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n), \quad \Leftarrow \text{faux!!} \end{aligned}$$

puisque c est une constante. L'erreur réside dans le fait que nous n'avons pas démontré la *forme exacte* de l'hypothèse de récurrence, à savoir que $T(n) \leq cn$.

e) Changement de variable

Parfois, il suffit d'une petite manipulation algébrique pour faire ressembler une récurrence inconnue à une autre déjà vue. Considérons, par exemple, la récurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

qui semble difficile. Il est pourtant possible de la simplifier *via* un changement de variables. Pour se faciliter la tâche, on ne se préoccupera pas d'arrondir les valeurs

comme \sqrt{n} à l'entier le plus proche. Si l'on fait le changement de variable $m = \lg n$, on obtient

$$T(2^m) = 2T(2^{m/2}) + m .$$

Si l'on pose maintenant $S(m) = T(2^m)$, on obtient la nouvelle récurrence

$$S(m) = 2S(m/2) + m ,$$

qui ressemble beaucoup à la récurrence (4.4). Effectivement, la nouvelle récurrence a la même solution : $S(m) = O(m \lg m)$. En repassant de $S(m)$ à $T(n)$, on obtient $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Exercices

4.1.1 Montrer que la solution de $T(n) = T(\lceil n/2 \rceil) + 1$ est $O(\lg n)$.

4.1.2 On a vu que la solution de $T(n) = 2T(\lfloor n/2 \rfloor) + n$ est $O(n \lg n)$. Montrer que la solution de cette récurrence est aussi $\Omega(n \lg n)$. En conclure que la solution est $\Theta(n \lg n)$.

4.1.3 Montrer que, en faisant une hypothèse de récurrence différente, on peut contourner la difficulté posée par la condition aux limites $T(1) = 1$ pour la récurrence (4.4) sans modifier les conditions aux limites pour la démonstration par récurrence.

4.1.4 Montrer que $\Theta(n \lg n)$ est la solution de la récurrence « exacte » (4.2) du tri par fusion.

4.1.5 Montrer que la solution de $T(n) = 2T(\lfloor n/2 \rfloor) + 17) + n$ est $O(n \lg n)$.

4.1.6 Résoudre la récurrence $T(n) = 2T(\sqrt{n}) + 1$ en effectuant un changement de variable. Votre solution doit être asymptotiquement approchée. Il ne faut pas se soucier de savoir si les valeurs sont entières.

4.2 MÉTHODE DE L'ARBRE RÉCURSIF

La méthode de substitution peut fournir une preuve succincte de la validité d'une solution pour une récurrence, mais il est parfois ardu d'imaginer une bonne conjecture. Tracer un arbre récursif, ainsi que nous l'avons fait dans notre analyse de la récurrence associée au tri par fusion (voir section 2.3.2), est un moyen direct d'arriver à une bonne conjecture. Dans un *arbre récursif*, chaque nœud représente le coût d'un sous-problème individuel, situé quelque part dans l'ensemble des invocations récursives de fonction. On totalise les coûts pour chaque niveau de l'arbre afin d'obtenir un ensemble de coûts par niveau, puis l'on cumule tous les coûts par niveau pour calculer le coût total de la récursivité tous niveaux confondus. Les arbres récursifs sont particulièrement utiles quand la récurrence décrit le temps d'exécution d'un algorithme diviser-pour-régner.

Un arbre récursif s'utilise de préférence pour générer une bonne conjecture, confirmée ensuite via la méthode de substitution. Quand on emploie un arbre récursif pour générer une bonne conjecture, on peut souvent tolérer une petite dose d'imprécision puisque l'on vérifiera ultérieurement la conjecture. Néanmoins, si l'on trace l'arbre récursif et que l'on cumule les coûts avec une très grande méticulosité, l'arbre fournit une preuve directe d'une solution pour la récurrence. Dans cette section, nous utiliserons des arbres récursifs pour produire de bonnes conjectures, sachant qu'à la section 4.4 nous emploierons directement les arbres récursifs pour démontrer le théorème sur lequel s'appuie la méthode générale.

Voyons, par exemple, comment un arbre récursif fournirait une bonne conjecture pour la récurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Commençons par nous occuper de trouver une borne supérieure pour la solution. Comme nous savons que les parties entières n'entrent généralement pas en ligne de compte pour la résolution des récurrences (voici un exemple d'approximation licite), nous créons un arbre récursif pour la récurrence $T(n) = 3T(n/4) + cn^2$, dans laquelle nous avons explicité le coefficient constant $c > 0$ implicite.

La figure 4.1 montre le tracé progressif de l'arbre récursif associé à $T(n) = 3T(n/4) + cn^2$. Pour des raisons de commodité, l'on suppose que n est une puissance exacte de 4 (autre exemple d'approximation tolérable). La partie (a) de la figure montre $T(n)$, qui est ensuite développé en partie (b) pour devenir un arbre équivalent représentant la récurrence. Le terme cn^2 situé sur la racine représente le coût au niveau supérieur de la récursivité, et les trois sous-arbres de la racine représentent les coûts induits par les sous-problèmes de taille $n/4$. La partie (c) montre le processus une étape plus loin, après expansion de chacun des nœuds de coût $T(n/4)$ de la partie (b). Le coût de chacun des trois enfants de la racine est $c(n/4)^2$. On continue de développer chaque nœud de l'arbre en le décomposant en ses parties constitutives, telles que déterminées par la récurrence.

Comme la taille du sous-problème décroît à mesure que l'on s'éloigne de la racine, on finira par atteindre une condition aux limites. À quelle distance de la racine cela se produira-t-il ? La taille de sous-problème pour un nœud situé à la profondeur i est de $n/4^i$. Donc, la taille de sous-problème prendra la valeur $n = 1$ quand $n/4^i = 1$ ou, de manière équivalente, quand $i = \log_4 n$. Par conséquent, l'arbre a $\log_4 n + 1$ niveaux $(0, 1, 2, \dots, \log_4 n)$.

Ensuite, on détermine le coût pour chaque niveau de l'arbre. Chaque niveau a trois fois plus de nœuds que le niveau immédiatement supérieur, de sorte que le nombre de nœuds de profondeur i est 3^i . Comme la taille du sous-problème diminue d'un facteur 4 quand on descend d'un niveau, chaque nœud de profondeur i , pour $i = 0, 1, 2, \dots, \log_4 n - 1$, possède un coût de $c(n/4^i)^2$. En multipliant, on voit que le coût total pour l'ensemble des nœuds de profondeur i , pour $i = 0, 1, 2, \dots, \log_4 n - 1$, vaut $3^i c(n/4^i)^2 = (3/16)^i cn^2$. Le dernier niveau, situé à la profondeur $\log_4 n$, a $3^{\log_4 n} = n^{\log_4 3}$ nœuds, qui ont chacun un coût $T(1)$ et qui donnent un coût total de $n^{\log_4 3} T(1)$, qui est $\Theta(n^{\log_4 3})$.

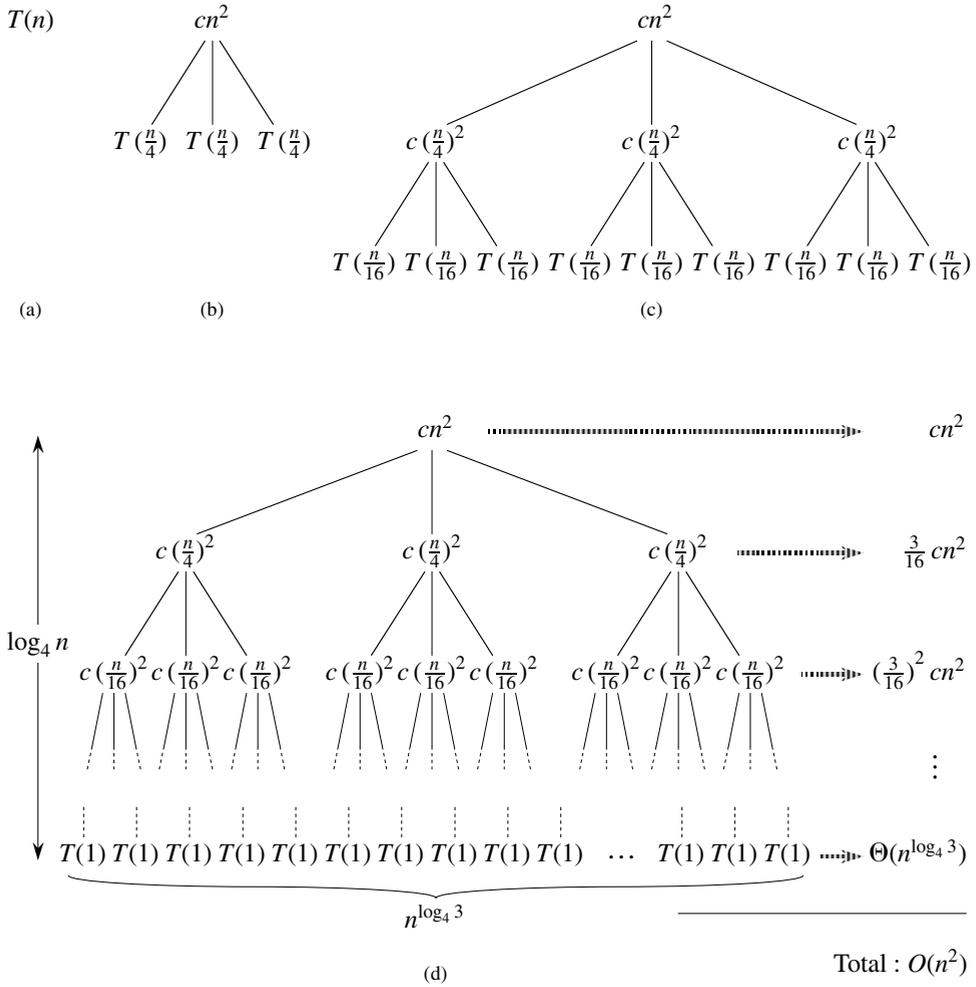


Figure 4.1 La construction d'un arbre récursif pour la récurrence $T(n) = 3T(n/4) + cn^2$. La partie (a) montre $T(n)$, progressivement développé dans les parties (b)–(d) pour former l'arbre récursif. L'arbre entièrement développé, en partie (d), a une hauteur de $\log_4 n$ (il comprend $\log_4 n + 1$ niveaux).

On cumule maintenant les coûts de tous les niveaux, afin de calculer le coût global de l'arbre :

$$\begin{aligned}
 T(n) &= cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}).
 \end{aligned}$$

Cette dernière formule paraît quelque peu tarabiscotée mais on peut, ici Aussi, utiliser une petite dose d'approximation pour borner supérieurement le résultat à l'aide d'une série géométrique décroissante. En reprenant l'avant-dernière formule et en appliquant l'équation (A.6), on obtient

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2). \end{aligned}$$

Nous avons donc déterminé une conjecture $T(n) = O(n^2)$ pour la récurrence originelle $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Dans cet exemple, les coefficients de cn^2 forment une série géométrique décroissante ; d'après l'équation (A.6), la somme de ces coefficients est bornée supérieurement par la constante $16/13$. Comme la contribution de la racine au coût global est cn^2 , la racine contribue pour une fraction constante du coût global. En d'autres termes, le coût total de l'arbre est dominé par le coût de la racine.

En fait, si $O(n^2)$ est bien une borne supérieure pour la récurrence (comme nous le vérifierons dans un moment), alors il doit s'agir d'une borne très approchée. Pourquoi donc ? Le premier appel récursif contribue pour un coût de $\Theta(n^2)$, et donc $\Omega(n^2)$ doit être une borne inférieure pour la récurrence.

On peut maintenant employer la méthode de substitution pour vérifier la validité de la conjecture, à savoir que $T(n) = O(n^2)$ est une borne supérieure de la récurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. On veut montrer que $T(n) \leq dn^2$ pour une certaine constante $d > 0$. En utilisant la même constante $c > 0$ que précédemment, on a

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 = \frac{3}{16} dn^2 + cn^2 \leq dn^2, \end{aligned}$$

la dernière étape étant vraie si $d \geq (16/13)c$.

Autre exemple, plus complexe, celui de la figure 4.2 qui montre l'arbre récursif de

$$T(n) = T(n/3) + T(2n/3) + O(n).$$

(Ici aussi, on omet les parties entières à des fins de simplification.) Comme précédemment, c représente le facteur constant dans le terme $O(n)$. Quand on cumule les valeurs à travers les niveaux de l'arbre, on obtient une valeur cn pour chaque niveau. Le plus long chemin menant de la racine à une feuille est $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \dots \rightarrow 1$. Comme $(2/3)^k n = 1$ quand $k = \log_{3/2} n$, la hauteur de l'arbre est $\log_{3/2} n$.

Intuitivement, on s'attend à ce que la solution de la récurrence soit au plus égale au nombre de niveaux multiplié par le coût de chaque niveau, soit $O(cn \log_{3/2} n) = O(n \lg n)$. Le coût total est uniformément distribué à travers les niveaux de l'arbre. Mais il y a une complication ici : il faut aussi considérer le coût

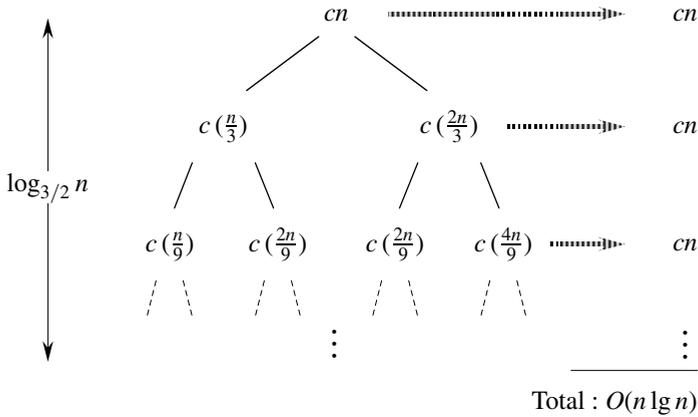


Figure 4.2 Un arbre récursif pour la récurrence $T(n) = T(n/3) + T(2n/3) + cn$.

des feuilles. Si l'arbre était un arbre binaire complet de hauteur $\log_{3/2} n$, il y aurait $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ feuilles. Comme le coût de chaque feuille est une constante, le coût cumulé de toutes les feuilles serait alors $\Theta(n^{\log_{3/2} 2})$, qui est $\omega(n \lg n)$. Mais cet arbre récursif n'est pas un arbre binaire complet, et donc il a moins de $n^{\log_{3/2} 2}$ feuilles. En outre, plus on s'éloigne de la racine, plus il y a de nœuds internes absents. Par conséquent, tous les niveaux ne contribuent pas pour un coût de cn ; les niveaux inférieurs contribuent pour une valeur moindre. On pourrait comptabiliser très précisément tous les coûts, mais rappelons-nous que l'on essaie tout simplement d'arriver à une conjecture qui soit utilisable dans la méthode de substitution. Tolérons donc l'approximation et essayons de montrer qu'une conjecture de $O(n \lg n)$ pour la borne supérieure est correcte.

Effectivement, on peut utiliser la méthode de substitution pour vérifier que $O(n \lg n)$ est une borne supérieure pour la solution de la récurrence. On montre que $T(n) \leq dn \lg n$, où d est une constante positive idoine. On a

$$\begin{aligned}
 T(n) &\leq T(n/3) + T(2n/3) + cn \\
 &\leq d(n/3) \lg(n/3) + d(2n/3) \lg(2n/3) + cn \\
 &= (d(n/3) \lg n - d(n/3) \lg 3) \\
 &\quad + (d(2n/3) \lg n - d(2n/3) \lg(3/2)) + cn \\
 &= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg(3/2)) + cn \\
 &= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg 3 - (2n/3) \lg 2) + cn \\
 &= dn \lg n - dn(\lg 3 - 2/3) + cn \\
 &\leq dn \lg n,
 \end{aligned}$$

pour $d \geq c/(\lg 3 - (2/3))$. Il n'était donc point nécessaire de faire un compte plus précis des coûts dans l'arbre récursif.

Exercices

4.2.1 Utiliser un arbre récursif pour déterminer une bonne borne supérieure asymptotique pour la récurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Vérifier la réponse à l'aide de la méthode de substitution.

4.2.2 Démontrer, en faisant appel à un arbre récursif, que la solution de la récurrence $T(n) = T(n/3) + T(2n/3) + cn$, où c est une constante, est $\Omega(n \lg n)$.

4.2.3 Dessiner l'arbre récursif de $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, où c est une constante, puis fournir une borne asymptotique approchée pour sa solution. Valider la borne à l'aide de la méthode de substitution.

4.2.4 Utiliser un arbre récursif pour donner une solution asymptotiquement approchée de la récurrence $T(n) = T(n - a) + T(a) + cn$, où $a \geq 1$ et $c > 0$ sont des constantes.

4.2.5 Utiliser un arbre récursif pour donner une solution asymptotiquement approchée de la récurrence $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, où α est une constante telle que $0 < \alpha < 1$ et $c > 0$ est une constante.

4.3 MÉTHODE GÉNÉRALE

La méthode générale donne une « recette » pour résoudre les récurrences de la forme

$$T(n) = aT(n/b) + f(n), \quad (4.5)$$

où $a \geq 1$ et $b > 1$ sont des constantes, et $f(n)$ une fonction asymptotiquement positive. La méthode générale oblige à traiter trois cas de figure, mais permet de déterminer la solution de nombreuses récurrences assez facilement.

La récurrence (4.5) décrit le temps d'exécution d'un algorithme qui divise un problème de taille n en a sous-problèmes, chacun de taille n/b , a et b étant des constantes positives. Les a sous-problèmes sont résolus récursivement, chacun dans un temps $T(n/b)$. Le coût induit par la décomposition du problème et la combinaison des résultats des sous-problèmes est décrit par la fonction $f(n)$. (C'est-à-dire par $f(n) = D(n) + C(n)$, si l'on reprend la notation vue à la section 2.3.2.) Ainsi, la récurrence résultant de la procédure TRI-FUSION donne $a = 2$, $b = 2$ et $f(n) = \Theta(n)$.

Rigoureusement parlant, la récurrence n'est pas vraiment bien définie, puisque n/b pourrait ne pas être un entier. Cependant, remplacer chacun des a termes $T(n/b)$ par $T(\lfloor n/b \rfloor)$ ou par $T(\lceil n/b \rceil)$ n'affecte pas le comportement asymptotique de la récurrence. (Cette affirmation sera démontrée à la prochaine section.) On trouve donc généralement commode d'omettre les parties entières quand on écrit des récurrences diviser-pour-régner de cette forme.

a) Théorème général

La méthode générale s'appuie sur le théorème suivant.

Théorème 4.1 (Théorème général.) Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers non négatifs par la récurrence

$$T(n) = aT(n/b) + f(n),$$

où l'on interprète n/b comme signifiant $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. $T(n)$ peut alors être bornée asymptotiquement de la façon suivante.

- 1) Si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- 2) Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \lg n)$.
- 3) Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Avant d'appliquer le théorème à des exemples, prenons un moment pour essayer de comprendre sa signification. Dans chacun des trois cas, on compare la fonction $f(n)$ à la fonction $n^{\log_b a}$. Intuitivement, la solution de la récurrence est déterminée par la plus grande des deux fonctions. Si c'est la fonction $n^{\log_b a}$ qui est la plus grande (cas 1), alors la solution est $T(n) = \Theta(n^{\log_b a})$. Si c'est la fonction $f(n)$ qui est la plus grande (cas 3), alors la solution est $T(n) = \Theta(f(n))$. Si les deux fonctions ont la même taille (cas 2), on multiplie par un facteur logarithmique et la solution est $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Derrière cette intuition se cachent certaines finesses techniques. Dans le premier cas, non seulement $f(n)$ doit être plus petite que $n^{\log_b a}$, mais elle doit être plus petite *polynomialement*. Autrement dit, $f(n)$ doit être asymptotiquement inférieure à $n^{\log_b a}$ d'un facteur n^ε , pour une certaine constante $\varepsilon > 0$. Dans le troisième cas, non seulement $f(n)$ doit être plus grande que $n^{\log_b a}$, mais elle doit être plus grande polynomialement et, en outre, satisfaire à la condition de « régularité » selon laquelle $af(n/b) \leq cf(n)$. Cette condition est satisfaite par la plupart des fonctions polynomialement bornées que nous rencontrerons.

Il faut bien comprendre que les trois cas ne recouvrent pas toutes les possibilités pour $f(n)$. Il y a un fossé entre les cas 1 et 2, quand $f(n)$ est plus petite que $n^{\log_b a}$ mais pas polynomialement plus petite. De même, il existe un fossé entre les cas 2 et 3 quand $f(n)$ est plus grande que $n^{\log_b a}$ mais pas polynomialement plus grande. Si la fonction $f(n)$ tombe dans l'un de ces fossés, ou si la condition de régularité du cas 3 n'est pas vérifiée, alors la méthode générale ne peut pas servir à résoudre la récurrence.

b) Utilisation de la méthode générale

Pour utiliser la méthode générale, on se contente de déterminer le cas du théorème général qui s'applique (s'il y en a un) puis d'écrire la réponse.

Comme premier exemple, considérons

$$T(n) = 9T(n/3) + n.$$

Pour cette récurrence, on a $a = 9$, $b = 3$ et $f(n) = n$; on a donc $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Puisque $f(n) = O(n^{\log_3 9 - \varepsilon})$, où $\varepsilon = 1$, on peut appliquer le cas 1 du théorème général et dire que la solution est $T(n) = \Theta(n^2)$.

Considérons à présent

$$T(n) = T(2n/3) + 1,$$

où $a = 1$, $b = 3/2$, $f(n) = 1$ et $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. On est dans le cas 2 puisque $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, et donc la solution de la récurrence est $T(n) = \Theta(\lg n)$.

Pour la récurrence

$$T(n) = 3T(n/4) + n \lg n,$$

on a $a = 3$, $b = 4$, $f(n) = n \lg n$ et $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$. Puisque $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$, où $\varepsilon \approx 0,2$, on est dans le cas 3 si l'on peut montrer que la condition de régularité est vraie pour $f(n)$. Pour n suffisamment grand,

$$af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n),$$

avec $c = 3/4$. Par conséquent, d'après le cas 3, la solution de la récurrence est $T(n) = \Theta(n \lg n)$.

La méthode générale ne s'applique pas à la récurrence

$$T(n) = 2T(n/2) + n \lg n,$$

bien qu'elle ait la forme correcte : $a = 2$, $b = 2$, $f(n) = n \lg n$ et $n^{\log_b a} = n$. On pourrait penser que le cas 3 s'applique, puisque $f(n) = n \lg n$ est plus grande asymptotiquement que $n^{\log_b a} = n$. Le problème est qu'elle n'est pas *polynomialement* plus grande. Le rapport $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ est asymptotiquement plus petit que n^ε pour toute constante positive ε . En conséquence, la récurrence tombe dans le fossé situé entre le cas 2 et le cas 3. (Voir exercice 4.4.2 pour une solution.)

Exercices

4.3.1 Utiliser la méthode générale pour donner des bornes asymptotiques approchées pour les récurrences suivantes.

- $T(n) = 4T(n/2) + n.$
- $T(n) = 4T(n/2) + n^2.$
- $T(n) = 4T(n/2) + n^3.$

4.3.2 Le temps d'exécution d'un algorithme A est décrit par la récurrence $T(n) = 7T(n/2) + n^2$. Un algorithme concurrent A' a un temps d'exécution décrit par $T'(n) = aT'(n/4) + n^2$. Quelle est la plus grande valeur entière de a telle que A' soit asymptotiquement plus rapide que A ?

4.3.3 Utiliser la méthode générale pour montrer que la solution de la récurrence $T(n) = T(n/2) + \Theta(1)$ associée à la recherche dichotomique) est $T(n) = \Theta(\lg n)$. (Voir exercice 2.3.5 pour une définition de la recherche dichotomique.)

4.3.4 La méthode générale est-elle applicable à la récurrence $T(n) = 4T(n/2) + n^2 \lg n$? Pourquoi? Donner une borne supérieure asymptotique pour cette récurrence.

4.3.5 * Considérez la condition de régularité $af(n/b) \leq cf(n)$, pour une certaine constante $c < 1$, qui fait partie du cas 3 du théorème général. Donner un exemple de constantes $a \geq 1$ et $b > 1$ et de fonction $f(n)$ qui satisfasse à toutes les conditions du cas 3 du théorème général, sauf à la condition de régularité.

4.4^{*} DÉMONSTRATION DU THÉORÈME GÉNÉRAL

Cette section contient une démonstration du théorème général (théorème 4.1). Il n'est pas nécessaire de comprendre la démonstration pour appliquer le théorème.

La preuve est divisée en deux parties. La première fait l'analyse de la récurrence « générale » (4.5), en faisant l'hypothèse simplificatrice que $T(n)$ n'est défini que sur les puissances exactes de $b > 1$, autrement dit, pour $n = 1, b, b^2, \dots$. Cette partie donne toute l'intuition nécessaire pour comprendre pourquoi le théorème général est vrai. La seconde partie montre comment l'analyse peut être étendue pour tous les entiers positifs n et est tout simplement une technique mathématique appliquée à la gestion des parties entières.

Dans cette section, nous abuserons parfois légèrement de notre notation asymptotique en l'employant pour décrire le comportement de fonctions qui ne sont définies que pour des puissances exactes de b . Il faut se souvenir que les définitions des notations asymptotiques imposent que les bornes soit démontrées pour tous les nombres suffisamment grands, et pas seulement pour ceux qui sont puissances de b . Comme on pourrait construire de nouvelles notations asymptotiques qui s'appliquent à l'ensemble $\{b^i : i = 0, 1, \dots\}$, au lieu des entiers positifs, l'abus est mineur.

Néanmoins, on doit toujours rester sur ses gardes lorsqu'on utilise les notations asymptotiques sur un domaine limité, de façon à ne pas tirer de conclusions hâtives. Par exemple, démontrer que $T(n) = O(n)$ quand n est une puissance exacte de 2 ne garantit pas que $T(n) = O(n)$. La fonction $T(n)$ pourrait être définie par

$$T(n) = \begin{cases} n & \text{si } n = 1, 2, 4, 8, \dots, \\ n^2 & \text{sinon,} \end{cases}$$

auquel cas la meilleure borne supérieure qu'on puisse trouver est $T(n) = O(n^2)$. À cause de ce type de conséquences fâcheuses, on n'utilisera jamais la notation asymptotique sur un domaine limité sans faire en sorte que cette utilisation soit rendue parfaitement claire par le contexte.

4.4.1 Démonstration pour les puissances exactes

La première partie de la preuve du théorème général analyse la récurrence (4.5),

$$T(n) = aT(n/b) + f(n) ,$$

de la méthode générale en faisant l'hypothèse que n est une puissance exacte de $b > 1$, où b n'est pas nécessairement entier. L'analyse se divise en trois lemmes. Le premier réduit le problème de résolution de la récurrence générale à un problème d'évaluation d'une expression qui contient une sommation. Le deuxième détermine les bornes de cette sommation. Le troisième lemme rassemble les deux premiers pour prouver une version du théorème général, dans le cas où n est une puissance exacte de b .

Lemme 4.2 Soient $a \geq 1$ et $b > 1$ deux constantes, et soit $f(n)$ une fonction non négative définie sur des puissances exactes de b . On définit $T(n)$ pour des puissances exactes de b par la récurrence

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 , \\ aT(n/b) + f(n) & \text{si } n = b^i , \end{cases}$$

où i est un entier positif. Alors :

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) . \quad (4.6)$$

Démonstration : Nous utiliserons l'arbre de la figure 4.3. La racine de l'arbre a le coût $f(n)$ et elle a a enfants, dont chacun a un coût $f(n/b)$. (Il est commode de se représenter a comme un entier, surtout quand on visualise l'arbre, mais mathématiquement cela n'est pas une obligation.) Chacun de ces enfants a a enfants de coût $f(n/b^2)$, et il y a donc a^2 nœuds à la distance 2 de la racine. Plus généralement, il y a a^j nœuds à la distance j de la racine, chacun ayant un coût $f(n/b^j)$. Le coût de chaque feuille est $T(1) = \Theta(1)$, et chaque feuille est à la profondeur $\log_b n$, vu que $n/b^{\log_b n} = 1$. Il y a $a^{\log_b n} = n^{\log_b a}$ feuilles dans l'arbre.

On peut obtenir l'équation (4.6) en cumulant les coûts de chaque niveau de l'arbre, comme le montre la figure. Le coût pour un niveau j de nœuds internes est $a^j f(n/b^j)$, et donc le coût total de tous les niveaux internes de nœuds est

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) .$$

Dans l'algorithme diviser-pour-régner sous-jacent, ce cumul représente les coûts induits par la décompositions des problèmes en sous-problèmes et par le regroupement ultérieur des sous-problèmes. Le coût de toutes les feuilles, qui est le coût cumulé de tous les $n^{\log_b a}$ sous-problèmes de taille 1, est $\Theta(n^{\log_b a})$. \square

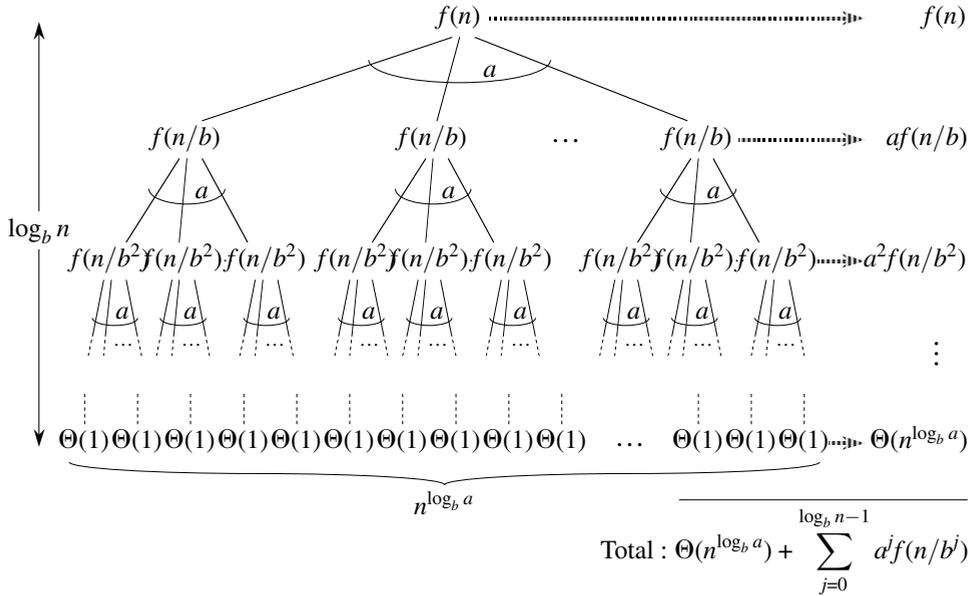


Figure 4.3 L'arbre récursif généré par $T(n) = aT(n/b) + f(n)$. L'arbre est un arbre complet d'arité a ayant $n^{\log_b a}$ feuilles et une hauteur $\log_b n$. Le coût de chaque niveau est montré à droite, et leur somme est donnée dans l'équation (4.6).

En termes d'arbre récursif, les trois cas du théorème général correspondent aux cas où le coût total de l'arbre est (1) dominé par les coûts des feuilles, (2) uniformément distribué à travers les niveaux de l'arbre ou (3) dominé par le coût de la racine.

Dans l'équation (4.6), la sommation décrit le coût des étapes diviser et combiner de l'algorithme diviser-pour-régner sous-jacent. Le lemme suivant donne des bornes asymptotiques pour la croissance de la sommation.

Lemme 4.3 Soient $a \geq 1$ et $b > 1$ deux constantes, et soit $f(n)$ une fonction positive définie pour des puissances exactes de b . Une fonction $g(n)$ définie pour des puissances exactes de b par

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{4.7}$$

peut être bornée asymptotiquement pour des puissances exactes de b de la manière suivante :

- 1) Si $f(n) = O(n^{\log_b a - \epsilon})$ pour une certaine constante $\epsilon > 0$, alors $g(n) = O(n^{\log_b a})$.
- 2) Si $f(n) = \Theta(n^{\log_b a})$, alors $g(n) = \Theta(n^{\log_b a} \lg n)$.
- 3) Si $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et tous $n \geq b$, alors $g(n) = \Theta(f(n))$.

Démonstration : Dans le cas 1, on a $f(n) = O(n^{\log_b a - \varepsilon})$, ce qui donne $f(n/b^j) = O((n/b^j)^{\log_b a - \varepsilon})$. Après substitution dans l'équation (4.7), on obtient :

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon}\right). \quad (4.8)$$

On borne l'équation à l'intérieur d'une notation O en factorisant les termes et en simplifiant, ce qui donne une série géométrique croissante :

$$\begin{aligned} \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon} &= n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} (b^\varepsilon)^j \\ &= n^{\log_b a - \varepsilon} \left(\frac{b^\varepsilon \log_b n - 1}{b^\varepsilon - 1}\right) \\ &= n^{\log_b a - \varepsilon} \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right). \end{aligned}$$

Puisque b et ε sont des constantes, la dernière expression se réduit à

$$n^{\log_b a - \varepsilon} O(n^\varepsilon) = O(n^{\log_b a}).$$

Après remplacement de la sommation par cette expression dans l'équation (4.8), on obtient

$$g(n) = O(n^{\log_b a}),$$

et le cas 1 est prouvé.

En supposant que $f(n) = \Theta(n^{\log_b a})$ pour le cas 2, on a $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. En substituant dans l'équation (4.7), on obtient :

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right). \quad (4.9)$$

Comme dans le cas 1, on borne la sommation avec Θ , mais cette fois, on n'obtient pas une série géométrique. On découvre en revanche que chaque terme de la sommation est le même :

$$\begin{aligned} \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1 \\ &= n^{\log_b a} \log_b n. \end{aligned}$$

En remplaçant par cette expression la sommation de l'équation (4.9), on obtient :

$$g(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n),$$

et le cas 2 est prouvé.

On démontre le cas 3 de la même manière. Puisque $f(n)$ apparaît dans la définition (4.7) de $g(n)$ et que tous les termes de $g(n)$ sont non négatifs, on peut conclure que $g(n) = \Omega(f(n))$ pour des puissances exactes de b . D'après notre hypothèse selon laquelle $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et tous les $n \geq b$, on a $a^j f(n/b^j) \leq c^j f(n)$. En itérant j fois, on a $f(n/b^j) \leq (c/a)^j f(n)$ ou, de manière équivalent, $a^j f(n/b^j) \leq c^j f(n)$. Après substitution dans l'équation (4.7) et simplification, on obtient une série géométrique, mais contrairement à celle du cas 1, celle-ci a des termes décroissants :

$$\begin{aligned} g(n) &\leq \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \\ &\leq \sum_{j=0}^{\log_b n - 1} c^j f(n) \\ &\leq f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n) \left(\frac{1}{1-c} \right) \\ &= O(f(n)), \end{aligned}$$

puisque c est constant. On peut donc conclure que $g(n) = \Theta(f(n))$ pour des puissances exactes de b . Le cas 3 est prouvé, ce qui termine la démonstration du lemme. \square

On peut prouver à présent une variante du théorème général pour le cas où n est une puissance exacte de b .

Lemme 4.4 Soient $a \geq 1$ et $b > 1$ des constantes, et soit $f(n)$ une fonction non négative définie pour des puissances exactes de b . On définit $T(n)$ pour des puissances exactes de b par la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ aT(n/b) + f(n) & \text{si } n = b^i, \end{cases}$$

où i est un entier positif. $T(n)$ peut alors être borné asymptotiquement pour des puissances exactes de b , de la manière suivante :

- 1) Si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- 2) Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \lg n)$.
- 3) Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.

Démonstration : On utilise les bornes du lemme 4.3 pour évaluer la sommation (4.6) à partir du lemme 4.2. Pour le cas 1, on a

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}), \end{aligned}$$

et pour le cas 2,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n). \end{aligned}$$

Pour le cas 3,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)), \end{aligned}$$

car $f(n) = \Omega(n^{\log_b a + \varepsilon})$. □

4.4.2 Parties entières

Pour compléter la démonstration du théorème général, il faut maintenant étendre notre analyse à la situation dans laquelle les parties entières sont utilisées dans la récurrence générale, pour que cette récurrence soit définie pour tous les entiers, et non plus uniquement pour les puissances exactes de b . L'obtention d'une borne inférieure pour

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

et d'une borne supérieure pour

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11}$$

se fait sans problème, puisque la borne $\lceil n/b \rceil \geq n/b$ peut être obtenue à partir du premier cas, et que la borne $\lfloor n/b \rfloor \leq n/b$ peut être obtenue grâce au deuxième cas. La technique permettant de borner par défaut la récurrence (4.11) est pratiquement identique à celle permettant de borner par excès la récurrence (4.10) ; on se contentera donc de prouver l'existence de cette dernière borne.

On modifie l'arbre récursif de la figure 4.3 pour produire l'arbre de la figure 4.4. À mesure que l'on descend dans l'arbre, on obtient une suite d'invocations récursives pour les arguments

$$\begin{aligned} n, \\ \lceil n/b \rceil, \\ \lceil \lceil n/b \rceil / b \rceil, \\ \lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil, \\ \vdots \end{aligned}$$

Notons n_i le i -ème élément de la séquence, où

$$n_i = \begin{cases} n & \text{si } i = 0, \\ \lceil n_{i-1}/b \rceil & \text{si } i > 0. \end{cases} \tag{4.12}$$

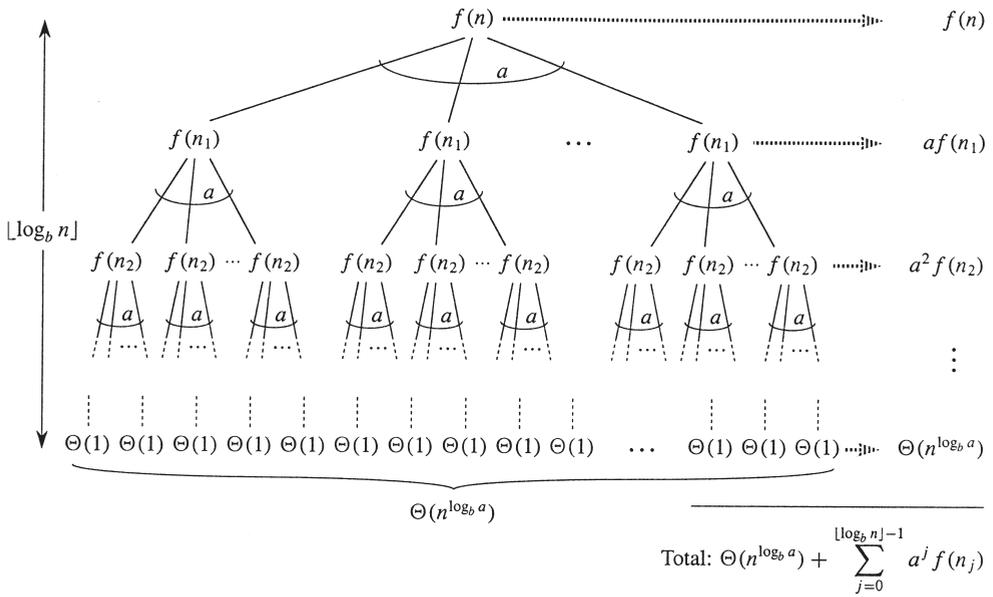


Figure 4.4 L'arbre récursif généré par $T(n) = aT(\lceil n/b \rceil) + f(n)$. L'argument récursif n_j est donné par l'équation (4.12).

Notre premier but est de déterminer le nombre d'itérations k telles que n_k soit une constante. A l'aide de l'inégalité $\lceil x \rceil \leq x + 1$, on obtient

$$\begin{aligned}
 n_0 &\leq n, \\
 n_1 &\leq \frac{n}{b} + 1, \\
 n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1, \\
 n_3 &\leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1, \\
 &\vdots
 \end{aligned}$$

D'une manière générale :

$$n_j \leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} < \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} = \frac{n}{b^j} + \frac{b}{b-1}.$$

En faisant $j = \lfloor \log_b n \rfloor$, on obtient

$$\begin{aligned}
 n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \leq \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\
 &= \frac{n}{n/b} + \frac{b}{b-1} = b + \frac{b}{b-1} = O(1),
 \end{aligned}$$

On voit donc qu'à la profondeur $\lfloor \log_b n \rfloor$, la taille de problème est au plus une constante. La figure 4.4 montre que

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j), \quad (4.13)$$

qui est pratiquement identique à l'équation (4.6), hormis que n est un entier arbitraire et qu'on ne lui impose pas d'être une puissance exacte de b .

Il est alors possible d'évaluer la sommation :

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \quad (4.14)$$

à partir de (4.13) de manière analogue à la démonstration du lemme 4.3. En commençant par le cas 3, si $af(\lceil n/b \rceil) \leq cf(n)$ pour $n > b + b/(b-1)$, où $c < 1$ est une constante, alors $a^j f(n_j) \leq c^j f(n)$. En conséquence, la somme dans l'équation (4.14) peut être évaluée exactement comme dans le lemme 4.3. Pour le cas 2, on a $f(n) = \Theta(n^{\log_b a})$. Si l'on peut montrer que $f(n_j) = O(n^{\log_b a} / a^j) = O((n/b^j)^{\log_b a})$, alors la preuve pour le cas 2 du lemme 4.3 suivra directement. Il faut observer que $j \leq \lfloor \log_b n \rfloor$ implique $b^j / n \leq 1$. La borne $f(n) = O(n^{\log_b a})$ implique qu'il existe une constante $c > 0$ telle que pour n_j suffisamment grand,

$$\begin{aligned} f(n_j) &\leq c \left(\frac{n}{b^j} + \frac{b}{b-1} \right)^{\log_b a} \\ &= c \left(\frac{n}{b^j} \left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1} \right) \right)^{\log_b a} \\ &= c \left(\frac{n^{\log_b a}}{a^j} \right) \left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1} \right) \right)^{\log_b a} \\ &\leq c \left(\frac{n^{\log_b a}}{a^j} \right) \left(1 + \frac{b}{b-1} \right)^{\log_b a} \\ &= O \left(\frac{n^{\log_b a}}{a^j} \right), \end{aligned}$$

puisque $c(1 + b/(b-1))^{\log_b a}$ est une constante. Le cas 2 est donc prouvé. La démonstration du cas 1 est pratiquement identique. Le principe est de démontrer que la borne $f(n_j) = O(n^{\log_b a - \varepsilon})$, ce qui équivaut à la démonstration correspondante dans le cas 2, bien que l'expression soit algébriquement plus complexe.

Nous avons maintenant démontré l'existence des bornes supérieures dans le théorème général pour tous les entiers n . La démonstration pour les bornes inférieures est similaire.

Exercices

4.4.1 ★ Donner une expression simple et exacte de n_i dans l'équation (4.12) pour le cas où b est un entier positif, et non un nombre réel arbitraire.

4.4.2 ★ Montrer que si $f(n) = \Theta(n^{\log_b a} \lg^k n)$, où $k \geq 0$, alors la récurrence générale a pour solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Pour simplifier, on restreindra l'analyse aux puissances exactes de b .

4.4.3 ★ Montrer que le cas 3 du théorème général est redondant, au sens où la condition de régularité $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ implique qu'il existe une constante $\varepsilon > 0$ telle que $f(n) = \Omega(n^{\log_b a + \varepsilon})$.

PROBLÈMES

4.1. Exemples de récurrences

Donner des bornes asymptotiques supérieure et inférieure pour $T(n)$ dans chacune des récurrences suivantes. On supposera que $T(n)$ est constant pour $n \leq 2$. Rendre les bornes aussi approchées que possible, et justifier les réponses.

- a. $T(n) = 2T(n/2) + n^3$.
- b. $T(n) = T(9n/10) + n$.
- c. $T(n) = 16T(n/4) + n^2$.
- d. $T(n) = 7T(n/3) + n^2$.
- e. $T(n) = 7T(n/2) + n^2$.
- f. $T(n) = 2T(n/4) + \sqrt{n}$.
- g. $T(n) = T(n-1) + n$.
- h. $T(n) = T(\sqrt{n}) + 1$.

4.2. Trouver l'entier manquant

Un tableau $A[1 \dots n]$ contient tous les entiers de 0 à n , sauf un. Il serait facile de déterminer l'entier manquant en un temps $O(n)$, à l'aide d'un tableau auxiliaire $B[0 \dots n]$ servant à ranger les nombres qui apparaissent dans A . Dans ce problème, toutefois, on ne peut pas accéder à un entier complet de A en une seule opération. Les éléments de A sont représentés en binaire, et la seule opération possible pour y accéder est « piocher le j -ème bit de $A[i]$ », ce qui prend un temps constant.

Montrer que, si l'on utilise uniquement cette opération, on peut encore déterminer l'entier manquant dans un temps $O(n)$.

4.3. Coût du passage de paramètres

Tout au long de ce livre, on suppose que le passage de paramètres lors d'un appel de procédure prend un temps constant, même si le paramètre est un tableau à N éléments. Cette supposition est valable sur la plupart des systèmes, car c'est un pointeur sur le tableau qui est passé, et non le tableau lui-même. Ce problème examine les conséquences de trois stratégies de passage de paramètre :

- 1) Un tableau est passé par pointeur. Temps = $\Theta(1)$.
 - 2) Un tableau est passé par copie. Temps = $\Theta(N)$, où N est la taille du tableau.
 - 3) Un tableau est passé via copie uniquement du sous-intervalle susceptible d'être utilisé par la procédure appelée. Temps = $\Theta(q - p + 1)$ si le sous-tableau $A[p..q]$ est passé.
- a. On considère l'algorithme récursif de recherche dichotomique consistant à trouver un nombre dans un tableau trié (voir exercice 2.3.5). Donner les récurrences correspondant aux temps d'exécution du cas le plus défavorable de la recherche dichotomique, pour chacune des trois méthodes précédentes de passage de tableau, et donner de bonnes bornes supérieures pour les solutions des récurrences. N est la taille du problème initial et n la taille d'un sous-problème.
- b. Refaire la partie (a) pour l'algorithme TRI-FUSION de la section 2.3.1.

4.4. Autres exemples de récurrences

Donner des bornes asymptotiques supérieure et inférieure pour $T(n)$ dans chacune des récurrences suivantes. On supposera que $T(n)$ est constant pour n suffisamment petit. Rendre les bornes aussi approchées que possible, et justifier les réponses.

- a. $T(n) = 3T(n/2) + n \lg n$.
- b. $T(n) = 5T(n/5) + n / \lg n$.
- c. $T(n) = 4T(n/2) + n^2 \sqrt{n}$.
- d. $T(n) = 3T(n/3 + 5) + n/2$.
- e. $T(n) = 2T(n/2) + n / \lg n$.
- f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.
- g. $T(n) = T(n - 1) + 1/n$.
- h. $T(n) = T(n - 1) + \lg n$.
- i. $T(n) = T(n - 2) + 2 \lg n$.
- j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4.5. Nombres de Fibonacci

Ce problème développe des propriétés des nombres de Fibonacci, qui sont définies par la récurrence (3.21). On utilisera la technique des fonctions génératrices pour

résoudre la récurrence de Fibonacci. On définit la *fonction génératrice* (ou *série formelle de puissances*) \mathcal{F} par :

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \dots,\end{aligned}$$

où F_i est le i -ème nombre de Fibonacci.

a. Montrer que $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Montrer que

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z} \right),\end{aligned}$$

où

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,61803\dots$$

et

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} = -0,61803\dots$$

c. Montrer que

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i) z^i.$$

d. Démontrer que $F_i = \phi^i / \sqrt{5}$ pour $i > 0$, arrondi à l'entier le plus proche. (*Conseil* : Observer que $|\hat{\phi}| < 1$.)

e. Démontrer que $F_{i+2} \geq \phi^i$ pour $i \geq 0$.

4.6. Test de puces VLSI

Le Professeur Diogène dispose de n puces VLSI⁽¹⁾, supposées identiques et théoriquement capables de se tester l'une l'autre. Le banc-test du professeur traite deux puces à la fois. Quand le banc est chargé, chaque puce teste l'autre et indique si elle est bonne ou mauvaise. Une bonne puce rapporte toujours correctement la qualité de l'autre puce, mais on ne peut pas se fier à la réponse d'une mauvaise puce. Les quatre résultats possibles pour un test sont donc :

(1) VLSI, acronyme de « very-large-scale integration » (intégration à très grande échelle) est la technologie de circuit intégré utilisée pour fabriquer la plupart des microprocesseurs actuels.

La puce A dit	La puce B dit	Conclusion
B est bonne	A est bonne	les deux sont bonnes, ou les deux sont mauvaises
B est bonne	A est mauvaise	l'une au moins est mauvaise
B est mauvaise	A est bonne	l'une au moins est mauvaise
B est mauvaise	A est mauvaise	l'une au moins est mauvaise

- Montrer que si plus de $n/2$ puces sont mauvaises, le professeur peut très bien ne pas être en mesure de déterminer quelles sont les bonnes puces, s'il s'appuie sur une stratégie fondée sur ce type de test de paire de puces. On supposera que les mauvaises puces sont capables de conspirer pour induire le professeur en erreur.
- On considère le problème consistant à trouver une bonne puce parmi n , en supposant que plus de $n/2$ puces sont bonnes. Montrer que $\lfloor n/2 \rfloor$ tests de paire de puces suffisent à réduire le problème de moitié environ.
- Montrer que les bonnes puces peuvent être identifiées avec $\Theta(n)$ tests de paire de puces, en supposant que plus de $n/2$ puces sont bonnes. Donner et résoudre la récurrence qui décrit le nombre de tests.

4.7. Tableaux de Monge

Un tableau A de réels de dimensions $m \times n$ est un **tableau de Monge** si, pour tout i, j, k et l vérifiant $1 \leq i < k \leq m$ et $1 \leq j < l \leq n$, on a

$$A[i, j] + A[k, l] \leq A[i, l] + A[k, j] .$$

En d'autres termes, chaque fois que l'on prend deux lignes et deux colonnes d'un tableau de Monge et que l'on regarde les quatre éléments situés aux intersections des lignes et des colonnes, la somme des éléments du coin supérieur gauche et du coin inférieur droit est inférieure ou égale à la somme des éléments du coin inférieur gauche et du coin supérieur droit. Voici un exemple de tableau de Monge :

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

- Montrer qu'un tableau est un tableau de Monge si et seulement si, pour tout $i = 1, 2, \dots, m - 1$ et tout $j = 1, 2, \dots, n - 1$, l'on a

$$A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j] .$$

(conseil : Pour la partie « si », utilisez la récurrence séparément sur les lignes et sur les colonnes.)

- b. Le tableau suivant n'est pas un tableau de Monge. Modifiez un élément pour en faire un tableau de Monge. (*Conseil* : Servez-vous de la partie (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

- c. Soit $f(i)$ l'indice de la colonne contenant l'élément minimal le plus à gauche de la ligne i . Montrer que $f(1) \leq f(2) \leq \dots \leq f(m)$ pour un quelconque tableau de Monge $m \times n$.
- d. Voici la description d'un algorithme diviser-pour-régner qui calcule l'élément minimal le plus à gauche pour chaque ligne d'un tableau de Monge $m \times n$ nommé A : Construire une sous-matrice A' of A composée des lignes de rang pair de A . Déterminer récursivement le minimum le plus à gauche pour chaque ligne de A' . Calculer ensuite le minimum le plus à gauche dans les lignes de rang impair de A . Expliquer comment calculer le minimum le plus à gauche dans les lignes de rang impair de A (à supposer que le minimum le plus à gauche des lignes de rang pair soit connu) dans un temps $O(m + n)$.
- e. Écrire la récurrence décrivant le temps d'exécution de l'algorithme décrit en partie (d). Montrer que sa solution est $O(m + n \log m)$.

NOTES

Les récurrences ont été étudiées dès 1202 par L. Fibonacci, qui a laissé son nom aux nombres de Fibonacci. A. De Moivre a introduit la méthode des fonctions génératrices (voir Problème 4.4.10) pour la résolution des récurrences. La méthode générale est adaptée d'après Bentley, Haken et Saxe [41], qui donne la méthode complète justifiée par l'exercice 4.4.4. Knuth [182] et Liu [205] montrent comment résoudre les récurrences linéaires à l'aide de la méthode des fonctions génératrices. Purdom et Brown [252], ainsi que Graham, Knuth et Patashnik [132] contiennent une vaste étude des récurrences.

Plusieurs chercheurs, dont Akra et Bazzi [13], Roura [262] et Verma [306] ont donné des méthodes pour résoudre des récurrences diviser-pour-régner plus générales que celles résolues par la méthode générale. Voici le résultat de Akra et Bazzi, obtenu pour les récurrences de la forme

$$T(n) = \sum_{i=1}^k a_i T(\lfloor n/b_i \rfloor) + f(n), \quad (4.15)$$

où $k \geq 1$; tous les coefficients a_i sont positifs et leur somme vaut au moins 1 ; tous les b_i sont supérieurs à 1 ; $f(n)$ est bornée, positive et non décroissante ; et, pour toute constante $c > 1$, il existe des constantes $n_0, d > 0$ telles que $f(n/c) \geq df(n)$ pour tout $n \geq n_0$. Cette méthode est utilisable pour une récurrence du genre $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, pour laquelle la méthode générale est inapplicable. Pour résoudre la récurrence (4.15), on commence par

trouver la valeur de p telle que $\sum_{i=1}^k a_i b_i^{-p} = 1$. (Un tel p existe toujours, et il est unique et positif.) La solution de la récurrence est alors

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^n \frac{f(x)}{x^{p+1}} dx\right),$$

où n' est une constante suffisamment grande. La méthode de Akra-Bazzi peut être quelque peu délicate d'utilisation, mais permet de résoudre des récurrences qui modélisent la division du problème en des sous-problèmes de tailles très inégales. La méthode générale est plus simple d'utilisation, mais n'est applicable que si les tailles des sous-problèmes sont égales.

Chapitre 5

Analyse probabiliste et algorithmes randomisés

Ce chapitre présente l'analyse probabiliste et les algorithmes randomisés. Si vous êtes novice en matière de probabilités, lisez l'annexe C qui rappelle les notions fondamentales. Analyse probabiliste et algorithmes randomisés reviendront à plusieurs reprises dans cet ouvrage.

5.1 LE PROBLÈME DE L'EMBAUCHE

Supposez que vous deviez embaucher une nouvelle secrétaire par le truchement d'une agence spécialisée. Cette agence vous envoie une candidate par jour. Vous interviewez cette personne pour savoir si vous allez l'embaucher ou non. L'agence prend des honoraires minimes pour chaque candidate qu'elle vous envoie, sachant qu'une embauche revient plus cher : en effet, vous devez congédier la secrétaire actuelle et payer des honoraires importants à l'agence. Comme vous avez décidé d'embaucher systématiquement la meilleure candidate, vous procédez de la manière suivante : après chaque interview, si la postulante est plus qualifiée que la secrétaire actuelle, vous renvoyez cette dernière pour engager la nouvelle candidate. Vous êtes prêt à payer le prix induit par cette stratégie, mais vous voulez avoir une idée de ce prix.

La procédure EMBAUCHE-SECRÉTAIRE, donnée ci-après, traduit en pseudo code cette stratégie d'embauche. Les candidates pour le poste sont ici numérotées de 1 à n . La procédure suppose que vous êtes capable, après avoir interviewé la candidate i , de déterminer si cette candidate i est la meilleure postulante que vous avez

vue jusqu'alors. À des fins d'initialisation, la procédure crée une candidate fictive, numérotée 0, qui est moins qualifiée que toutes les autres candidates.

EMBAUCHE-SECRÉTAIRE(n)

```

1  meilleure ← 0   ▷ candidate 0 est une candidate fictive, moins qualifiée
                        que quiconque
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire interviewer candidate  $i$ 
4          si candidate  $i$  supérieure à candidate meilleure
5              alors meilleure ←  $i$ 
6              embaucher candidate  $i$ 

```

Le modèle de coût de ce problème diffère du modèle présenté au chapitre 2. Nous ne nous intéressons pas au temps d'exécution de EMBAUCHE-SECRÉTAIRE, mais plutôt au coût induit par l'interview et l'embauche. À priori, analyser le coût de cet algorithme peut sembler très différent d'analyser le temps d'exécution d'une procédure telle que, disons, le tri par fusion. Les techniques analytiques employées sont néanmoins les mêmes, qu'il s'agisse d'analyser le coût ou le temps d'exécution. Dans l'un ou l'autre cas, nous comptons le nombre de fois que sont exécutées certaines opérations de base.

Une interview a un coût minime c_i , alors qu'une embauche a un coût c_h bien plus élevé. Soit m le nombre de personnes embauchées. Le coût total associé à cet algorithme est donc $O(nc_i + mc_h)$. Quel que soit le nombre de personnes que nous embauchons, nous interviewons toujours n candidates et ainsi nous avons toujours le même coût nc_i associé aux interviews. Nous allons, par conséquent, nous concentrer sur l'analyse de mc_h , le coût d'embauche. Cette quantité varie avec chaque exécution de l'algorithme.

Ce scénario sert de modèle à un paradigme informatique courant. Nous sommes fréquemment amenés à trouver la valeur maximale ou minimale d'une suite en examinant chaque élément de la suite et en gérant un « élu » courant. Le problème de l'embauche modélise le phénomène suivant : nous changeons plus ou moins souvent d'opinion quant à notre estimation de l'élément que nous pensons être le meilleur.

a) Analyse du cas le plus défavorable

Dans le cas le plus défavorable, nous embauchons chaque candidate interviewée. Cette situation se produit si les candidates se présentent dans l'ordre de qualité croissant, auquel cas nous embauchons n fois, pour un coût total de $O(nc_h)$.

Il est plus raisonnable, cependant, de s'attendre à ce que les candidates ne se présenteront pas systématiquement dans l'ordre de qualité croissant. En fait, nous n'avons aucune idée de l'ordre dans lequel elles viendront, et nous n'avons pas non plus de contrôle sur cet ordre. Il est donc naturel de nous demander ce que nous pensons qu'il adviendra dans un cas typique.

b) Analyse probabiliste

Par *analyse probabiliste*, l'on entend l'utilisation des probabilités pour analyser les problèmes. La plupart du temps, l'analyse probabiliste nous servira à étudier le temps d'exécution d'un algorithme. Parfois, nous l'utiliserons pour étudier d'autres valeurs, tel le coût d'embauche dans la procédure EMBAUCHE-SECRÉTAIRE. Pour effectuer une analyse probabiliste, nous devons nous baser sur la connaissance que nous avons de la distribution des données en entrée, ou alors faire des hypothèses sur cette distribution. Ensuite nous analysons notre algorithme, en calculant un temps d'exécution attendu. Comme l'espérance est prise sur la distribution des entrées possibles, nous faisons en réalité la moyenne des temps d'exécution sur toutes les entrées potentielles.

Il nous faut être très prudent quant à notre estimation concernant la distribution des entrées. Pour certains problèmes, il est raisonnable de faire des suppositions au sujet de l'ensemble de toutes les entrées potentielles ; nous pouvons alors employer l'analyse probabiliste comme technique de conception d'algorithme efficace et comme moyen de nous faire une meilleure idée d'un problème. Pour d'autres problèmes, en revanche, nous ne pouvons pas décrire de distribution des entrées raisonnable ; en pareil cas, nous ne pouvons pas nous servir de l'analyse probabiliste.

Pour ce qui concerne le problème de l'embauche, nous pouvons supposer que les postulantes se présentent dans un ordre aléatoire. Qu'est-ce que cela signifie ici ? Nous partons du principe que nous pouvons faire une comparaison entre deux candidates quelconques, puis décider quelle est la plus qualifiée ; il existe donc un ordre total pour les candidates. (Voir l'annexe B pour le concept d'ordre total.) Nous pouvons alors affecter à chaque candidate un numéro unique compris entre 1 et n , $\text{rang}(i)$ désignant ici le classement de la postulante i , et adopter la convention selon laquelle plus le rang est élevé et plus la candidate est qualifiée. La liste ordonnée $\langle \text{rang}(1), \text{rang}(2), \dots, \text{rang}(n) \rangle$ est une permutation de la liste $\langle 1, 2, \dots, n \rangle$. Dire que les candidates se présentent dans un ordre aléatoire revient à dire que la liste des classements peut être l'une quelconque des $n!$ permutations des nombres 1 à n . Nous pouvons aussi exprimer ce fait en disant que les classements constituent une *permutation aléatoire uniforme* ; cela veut dire que chacune des $n!$ permutations potentielles a la même probabilité d'apparition.

La section 5.2 contient une analyse probabiliste du problème de l'embauche.

c) Algorithmes randomisés

Pour pouvoir utiliser l'analyse probabiliste, nous devons connaître quelque chose concernant la distribution relative aux données en entrée. Bien souvent, nous ne savons pas grand chose sur la distribution des entrées. Et même si nous savons quelque chose, nous ne savons pas toujours modéliser informatiquement cette connaissance. Il n'empêche que nous pouvons souvent faire appel aux probabilités et aux modèles aléatoires pour nous aider à concevoir et analyser les algorithmes, et ce en contrôlant une partie des aléas de l'algorithme.

Dans le problème de l'embauche, on peut penser que les candidates nous sont envoyées dans un ordre aléatoire, mais nous n'avons aucun moyen de savoir si tel est bien le cas. Pour créer un algorithme randomisé pour le problème de l'embauche, nous devons avoir un meilleur contrôle sur l'ordre d'interview des candidates. Nous allons donc modifier légèrement le modèle. Nous allons supposer que l'agence nous fait parvenir à l'avance la liste des n candidates. Chaque jour, nous choisissons au hasard une candidate à interviewer. Nous ne connaissons rien des candidates (si ce n'est leurs noms), mais nous avons procédé à un changement de taille. Au lieu de supposer que les candidates viendront à nous dans un ordre aléatoire, nous contrôlons maintenant le processus pour imposer un ordre véritablement aléatoire.

Plus généralement, nous dirons qu'un algorithme est *randomisé* si son comportement dépend non seulement des données en entrée mais aussi de valeurs produites par un *générateur de nombres aléatoires*. Nous supposons que nous disposons d'un générateur de nombres aléatoires RANDOM. Un appel à RANDOM(a, b) donne un entier compris entre a et b inclus, chacun des entiers possibles ayant la même probabilité d'apparition. Ainsi, RANDOM(0, 1) donne 0 avec la probabilité $1/2$ et 1 avec la probabilité $1/2$. RANDOM(3, 7) donne 3, 4, 5, 6 ou 7, chacune de ces valeurs ayant la probabilité $1/5$. Chaque entier produit par RANDOM est indépendant des entiers produits par les appels précédents. Vous pouvez vous représenter RANDOM comme quelqu'un qui ferait rouler un dé à $(b - a + 1)$ faces. (Dans la pratique, la plupart des environnements de programmation offrent un générateur de nombres pseudo aléatoires, à savoir un algorithme déterministe produisant des nombres qui « ont l'air » statistiquement aléatoires.)

Exercices

5.1.1 Montrer que l'hypothèse selon laquelle nous sommes constamment en mesure de déterminer la candidate optimale sur la ligne 4 de la procédure EMBAUCHE-SECRETARE exige que nous connaissions un ordre total concernant les classements des candidates.

5.1.2 * Décrire une implémentation de la procédure RANDOM(a, b) qui ne fait que des appels à RANDOM(0, 1). Quel est le temps d'exécution attendu de votre procédure, en tant que fonction de a et b ?

5.1.3 * Supposez que vous vouliez produire 0 avec la probabilité $1/2$ et 1 avec la probabilité $1/2$. Vous disposez d'une procédure BIASED-RANDOM, qui donne 0 ou 1. Elle produit 1 avec une certaine probabilité p et 0 avec la probabilité $1 - p$ (avec $0 < p < 1$), mais vous ne connaissez pas la valeur de p . Donnez un algorithme utilisant BIASED-RANDOM comme sous-routine qui produise une réponse non faussée, c'est-à-dire qui produise 0 avec la probabilité $1/2$ et 1 avec la probabilité $1/2$. Quel est le temps d'exécution attendu de votre algorithme, en tant que fonction de p ?

5.2 VARIABLES INDICATRICES

Pour analyser moult algorithmes, dont le problème de l'embauche, nous utiliserons des variables aléatoires indicatrices. Les variables aléatoires indicatrices offrent une méthode commode pour faire la conversion entre probabilités et espérances. Soient S un espace d'épreuves et A un événement. La *variable indicatrice* $I\{A\}$ associée à l'événement aléatoire A est alors définie par

$$I\{A\} = \begin{cases} 1 & \text{si } A \text{ se produit,} \\ 0 & \text{si } A \text{ ne se produit pas.} \end{cases} \quad (5.1)$$

Comme exemple simple, calculons le nombre attendu de piles quand nous jetons en l'air une pièce non faussée. Notre espace d'épreuves est $S = \{P, F\}$, et nous définissons une variable aléatoire Y qui prend les valeurs P et F , dont chacune a la probabilité $1/2$. Nous pouvons alors définir une variable indicatrice X_P , associée au résultat pile, que nous pouvons exprimer comme étant l'événement $Y = P$. Cette variable compte le nombre de piles obtenu dans ce lancer ; c'est 1 si la pièce donne pile, 0 sinon. Nous écrivons

$$X_P = I\{Y = P\} = \begin{cases} 1 & \text{si } Y = P, \\ 0 & \text{si } Y = F. \end{cases}$$

Le nombre attendu de piles pour un seul lancer est tout simplement l'espérance de notre variable indicatrice X_P :

$$\begin{aligned} E[X_P] &= E[I\{Y = P\}] \\ &= 1 \cdot \Pr\{Y = P\} + 0 \cdot \Pr\{Y = F\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2. \end{aligned}$$

Le nombre attendu de piles dans un seul lancer d'une pièce non faussée est donc $1/2$. Ainsi que le montre le lemme suivant, l'espérance d'une variable indicatrice associée à un événement aléatoire A est égale à la probabilité de A .

Lemme 5.1 *Étant donné un espace d'épreuves S et un événement A de l'espace S , soit $X_A = I\{A\}$. Alors, $E[X_A] = \Pr\{A\}$.*

Démonstration : D'après la définition d'une variable indicatrice, telle que donnée dans l'équation (5.1), et d'après la définition de l'espérance, nous avons

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\bar{A}\} \\ &= \Pr\{A\}, \end{aligned}$$

\bar{A} désigne $S - A$, c'est-à-dire le complément de A . □

Les variables indicatrices peuvent paraître inutilement compliquées pour une application telle que le comptage du nombre attendu de piles pour un lancer d'une seule

pièce, mais elles s'avèrent précieuses pour analyser des situations dans lesquelles on effectue des essais aléatoires répétés. Par exemple, les variables indicatrices fournissent un moyen simple d'arriver au résultat de l'équation (C.36). Dans cette équation, on calcule le nombre de piles dans n lancers de pièce en considérant séparément la probabilité d'obtenir 0 pile, 1 pile, 2 piles, etc. Encore que la méthode plus simple proposée dans l'équation (C.37) utilise, en fait, implicitement des variables aléatoires indicatrices. Pour rendre cette discussion plus explicite, on peut définir X_i comme étant la variable indicatrice associée à l'événement par lequel le i -ème lancer donne pile. Si Y_i est la variable aléatoire désignant le résultat du i -ème lancer, on a $X_i = I\{Y_i = P\}$. Soit X la variable aléatoire désignant le nombre total de piles dans les n lancers, de sorte que

$$X = \sum_{i=1}^n X_i .$$

Comme on veut calculer le nombre attendu de piles, on prend l'espérance des deux côtés de l'équation précédente afin d'obtenir

$$E[X] = E \left[\sum_{i=1}^n X_i \right] .$$

Le côté gauche de l'équation qui précède est l'espérance de la somme de n variables aléatoires. D'après le lemme 5.1, nous pouvons facilement calculer l'espérance de chacune des variables aléatoires. D'après l'équation (C.2) —relative à la linéarité de l'espérance—, il est facile de calculer l'espérance de la somme : elle est égale à la somme des espérances des n variables aléatoires. La linéarité de l'espérance fait de l'emploi de variables indicatrices une puissante technique analytique ; elle s'applique même en cas de dépendances entre les variables aléatoires. Nous pouvons maintenant calculer facilement le nombre attendu de piles :

$$\begin{aligned} E[X] &= E \left[\sum_{i=1}^n X_i \right] \\ &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n 1/2 \\ &= n/2 . \end{aligned}$$

Ainsi, comparée à la méthode employée dans l'équation (C.36), les variables indicatrices simplifient grandement les calculs. Nous utiliserons des variables indicatrices tout au long de ce livre.

a) *Problème de l'embauche et variables indicatrices*

Revenons au problème de l'embauche ; nous voulons maintenant calculer le nombre attendu de fois que nous embauchons une nouvelle secrétaire. Pour pouvoir faire une analyse probabiliste, nous supposons que les candidates arrivent dans un ordre aléatoire, comme traité à la section précédente. (Nous verrons à la section 5.3 comment nous débarrasser de cette hypothèse). Soit X la variable aléatoire dont la valeur est égale au nombre de fois que nous engageons une secrétaire. Nous pourrions alors appliquer la définition de l'espérance, telle que donnée dans l'équation (C.19), pour obtenir

$$E[X] = \sum_{x=1}^n x \Pr\{X = x\} ,$$

mais ce calcul serait lourd. Nous utiliserons plutôt des variables indicatrices pour simplifier grandement le calcul.

Pour utiliser des variables indicatrices au lieu de calculer $E[X]$ en définissant une unique variable associée au nombre de fois que nous embauchons une secrétaire, nous définirons n variables indiquant si chaque postulante individuelle est embauchée ou non. En particulier, soit X_i la variable indicatrice associée à l'événement par lequel on engage la i -ème candidate. Alors,

$$X_i = I\{\text{candidate } i \text{ est embauchée}\} = \begin{cases} 1 & \text{si candidate } i \text{ est embauchée ,} \\ 0 & \text{si candidate } i \text{ n'est pas embauchée ,} \end{cases} \quad (5.2)$$

et

$$X = X_1 + X_2 + \cdots + X_n . \quad (5.3)$$

D'après le lemme 5.1, nous avons

$$E[X_i] = \Pr\{\text{candidate } i \text{ est embauchée}\} ,$$

et nous devons donc calculer la probabilité d'exécution des lignes 5–6 de EMBAUCHE-SECRÉTAIRE.

La candidate i est embauchée (ligne 5) si et seulement si la candidate i est meilleure que chacune des candidates 1 à $i - 1$. Comme nous sommes partis du principe que les candidates arrivent dans un ordre aléatoire, les i premières candidates se sont présentées dans un ordre aléatoire. Chacune de ces i premières candidates a la même probabilité de meilleure qualification provisoire. La candidate i a une probabilité $1/i$ d'être plus qualifiée que les candidates 1 à $i - 1$, et elle a donc une probabilité $1/i$ d'être engagée. D'après le lemme 5.1, on en conclut que

$$E[X_i] = 1/i . \quad (5.4)$$

Nous pouvons maintenant calculer $E[X]$:

$$E[X] = E\left[\sum_{i=1}^n X_i\right] \quad (\text{d'après équation (5.3)}) \quad (5.5)$$

$$= \sum_{i=1}^n E[X_i] \quad (\text{d'après linéarité de l'espérance})$$

$$= \sum_{i=1}^n 1/i \quad (\text{d'après équation (5.4)})$$

$$= \ln n + O(1) \quad (\text{d'après équation (A.7)}) . \quad (5.6)$$

Même si nous interviewons n personnes, en moyenne nous n'en embauchons réellement qu'environ $\ln n$. Nous résumerons ce résultat dans le lemme que voici.

Lemme 5.2 *Si les candidates se présentent dans un ordre aléatoire, l'algorithme EMBAUCHE-SECRÉTAIRE a un coût total d'embauche $O(c_h \ln n)$.*

Démonstration : La borne découle directement de notre définition du coût d'embauche et de l'équation (5.6). \square

Le coût d'embauche attendu représente une amélioration significative par rapport au coût d'embauche du cas le plus défavorable qui est $O(nc_h)$.

Exercices

5.2.1 Dans EMBAUCHE-SECRÉTAIRE, si les candidates se présentent dans un ordre aléatoire, quelle est la probabilité que vous n'embauchiez qu'une seule fois ? Quelle est la probabilité que vous embauchiez exactement n fois ?

5.2.2 Dans EMBAUCHE-SECRÉTAIRE, si les candidates se présentent dans un ordre aléatoire, quelle est la probabilité que vous embauchiez exactement deux fois ?

5.2.3 Utilisez des variables aléatoires indicatrices pour calculer l'espérance de la somme de n dés.

5.2.4 Utilisez des variables indicatrices pour résoudre le problème suivant, connu sous le nom de **problème du vestiaire à chapeaux**. Chaque client parmi n au total donne son chapeau à un employé d'un restaurant. Cet employé redonne les chapeaux aux clients dans un ordre aléatoire. Quel est le nombre attendu de clients qui récupéreront leurs chapeaux ?

5.2.5 Soit $A[1..n]$ un tableau de n nombres distincts. Si $i < j$ et $A[i] > A[j]$, on dit que la paire (i, j) est une **inversion** de A . (Voir problème 2.4 pour plus de détails sur les inversions.) On suppose que les éléments de A constituent une permutation uniforme aléatoire de l'intervalle 1 à n . Utilisez des variables indicatrices pour calculer le nombre attendu d'inversions.

5.3 ALGORITHMES RANDOMISÉS

Dans la section précédente, nous avons montré comment le fait de connaître une distribution concernant les entrées peut faciliter l'analyse du comportement moyen d'un algorithme. Bien souvent, nous n'avons pas une telle connaissance et nous ne pouvons faire aucune analyse de cas moyen. Comme mentionné à la section 5.1, nous pourrions peut-être employer un algorithme randomisé.

Pour un problème tel que celui de l'embauche, où cela aide de supposer que toutes les permutations de l'entrée sont équiprobables, une analyse probabiliste guide la création d'un algorithme randomisé. Au lieu de supposer qu'il y a telle ou telle distribution des entrées, nous imposons une distribution. En particulier, avant d'exécuter l'algorithme, nous permutons de manière aléatoire les candidates de façon à forcer l'équiprobabilité des diverses permutations. Cette modification ne change pas notre espérance d'embaucher une secrétaire environ $\ln n$ fois. Elle signifie, toutefois, que nous nous attendons à ce que cela soit le cas pour *chaque* entrée, et pas seulement pour des données tirées d'une distribution particulière.

Nous allons maintenant étudier plus avant la distinction entre analyse probabiliste et algorithmes randomisés. À la section 5.2, nous avons affirmé que, si les candidates se présentent dans un ordre aléatoire, le nombre attendu de fois que nous embauchons une secrétaire vaut environ $\ln n$. Notez que l'algorithme ici est déterministe ; pour une même entrée, quelle qu'elle soit, le nombre d'embauches d'une nouvelle secrétaire est toujours le même. En outre, le nombre de fois que nous engageons une secrétaire varie avec chaque entrée et dépend des classements des diverses candidates. Comme ce nombre ne dépend que des classements des candidates, nous pouvons représenter une entrée particulière par l'énumération ordonnée des classements des candidates, c'est-à-dire $\langle \text{rang}(1), \text{rang}(2), \dots, \text{rang}(n) \rangle$. Avec la liste $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$, il y aura toujours 10 embauches successives, vu que chaque candidate est meilleure que la précédente, et les lignes 5–6 seront exécutées pour chaque itération de l'algorithme. Avec la liste $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$, il n'y aura qu'une seule embauche, faite dans la première itération. Avec la liste $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$, il y aura trois embauches, associées aux candidates 5, 8 et 10. Si nous nous rappelons que le coût de notre algorithme dépend du nombre d'embauches, nous voyons qu'il existe des entrées coûteuses, telle A_1 , des entrées peu coûteuses, telle A_2 , et des entrées modérément coûteuses, telle A_3 .

Considérez, en revanche, l'algorithme randomisé qui commence par permuter les candidates avant de déterminer la meilleure. Ici, la randomisation se situe dans l'algorithme, pas dans la distribution des entrées. Étant donnée une certaine entrée, disons le A_3 précédent, on ne peut pas dire combien de fois il y aura actualisation du maximum, car cette valeur varie avec chaque exécution de l'algorithme. La première fois que nous exécutons l'algorithme sur A_3 , il peut générer la permutation A_1 et faire 10 actualisations ; la seconde fois que nous exécutons l'algorithme, nous pouvons produire la permutation A_2 et faire une seule mise à jour. La troisième fois que nous

l'exécutons, nous pouvons obtenir un autre nombre d'actualisations. À chaque exécution de l'algorithme, l'exécution dépend des choix aléatoires effectués et il y a de fortes chances pour qu'elle diffère de la précédente exécution. Pour cet algorithme, ainsi que pour nombre d'autres algorithmes randomisés, *il n'existe aucune entrée particulière qui engendre le comportement du cas le plus défavorable*. Même votre pire ennemi ne peut pas produire un tableau méchant en entrée, car la permutation aléatoire ôte toute importance à l'ordre des entrées. L'algorithme randomisé ne donne de mauvais résultats que si le générateur de nombres aléatoires génère une permutation « malheureuse ».

Pour le problème de l'embauche, la seule chose à modifier dans le code est la permutation aléatoire du tableau.

```

EMBAUCHE-SECRÉTAIRE-RANDOMISÉ( $n$ )
1  permuter aléatoirement la liste des candidates
2   $meilleure \leftarrow 0$   ▷ candidate 0 est une candidate fictive moins qualifiée
   que quiconque
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire interviewer candidate  $i$ 
5          si candidate  $i$  supérieure à candidate  $meilleure$ 
6              alors  $meilleure \leftarrow i$ 
7              embaucher candidate  $i$ 

```

Grâce à cette simple modification, nous avons créé un algorithme randomisé dont les performances correspondent à celles obtenues dans l'hypothèse où les candidates se présentent dans un ordre aléatoire.

Lemme 5.3 *Le coût d'embauche attendu de la procédure EMBAUCHE-SECRÉTAIRE-RANDOMISÉ est $O(c_h \ln n)$.*

Démonstration : Une fois permuté le tableau des entrées, nous sommes ramenés à une situation identique à celle de l'analyse probabiliste de EMBAUCHE-SECRÉTAIRE. □

La comparaison entre les lemmes 5.2 et 5.3 illustre la différence entre analyse probabiliste et algorithmes randomisés. Dans le lemme 5.2, nous faisons une hypothèse concernant l'entrée. Dans le lemme 5.3, nous ne faisons aucune hypothèse de ce genre, bien que la randomisation de l'entrée demande du temps supplémentaire. Dans le reste de cette section, nous allons étudier certains problèmes induits par la permutation aléatoire des entrées.

a) Permutation aléatoire de tableau

Nombreux sont les algorithmes randomisés qui randomisent l'entrée en permutant le tableau initial. (Il existe d'autres façons d'employer la randomisation.) Nous allons ici présenter deux méthodes en ce sens. Nous supposons que nous avons un tableau A

contenant les éléments 1 à n (cela ne nuit pas à la généralité du problème). Notre but est de produire une permutation aléatoire du tableau.

Une méthode courante consiste à assigner à chaque élément $A[i]$ du tableau une priorité aléatoire $P[i]$, puis à trier les éléments de A selon ces priorités. Par exemple, si notre tableau initial est $A = \langle 1, 2, 3, 4 \rangle$ et que nous choisissons les priorités aléatoires $P = \langle 36, 3, 97, 19 \rangle$, nous produirons un tableau $B = \langle 2, 4, 1, 3 \rangle$; en effet, la deuxième priorité est la plus faible, suivie de la quatrième, puis de la première, puis enfin de la troisième. Nous baptiserons cette procédure PERMUTE-PAR-TRI :

```

PERMUTE-PAR-TRI(A)
1   $n \leftarrow \text{longueur}[A]$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire  $P[i] = \text{RANDOM}(1, n^3)$ 
4  trier  $A$ , avec  $P$  comme clés de tri
5  retourner  $A$ 

```

Sur la ligne 3, il y a sélection d'un nombre aléatoire compris entre 1 et n^3 . Nous utilisons la plage 1 à n^3 pour augmenter les chances d'unicité de chaque priorité de P . (L'exercice 5.3.5 vous demandera de prouver que la probabilité que toutes les priorités soient uniques est au moins de $1 - 1/n$, alors que l'exercice 5.3.6 vous demandera comment implémenter l'algorithme s'il existe deux ou plusieurs priorités identiques.) Supposons ici que toutes les priorités sont uniques.

La phase qui prend du temps dans cette procédure, c'est le tri sur la ligne 4. Comme nous le verrons au chapitre 8, si nous employons un tri par comparaison, la durée du tri est de $\Omega(n \lg n)$. Nous pouvons arriver à cette limite inférieure, car nous avons vu que le tri par fusion prend $\Theta(n \lg n)$. (Nous verrons, dans la partie 2, d'autres tris par comparaison qui prennent $\Theta(n \lg n)$). Après le tri, si $P[i]$ est la j -ème priorité en partant de la valeur minimale, alors $A[i]$ est en position j dans le résultat. De cette manière, nous obtenons une permutation. Reste à prouver que la procédure donne une **permutation aléatoire uniforme**, c'est-à-dire que chaque permutation des nombres 1 à n a la même probabilité de génération.

Lemme 5.4 *La procédure PERMUTE-PAR-TRI produit une permutation aléatoire uniforme de l'entrée, dans l'hypothèse où toutes les priorités sont distinctes.*

Démonstration : Nous commencerons par considérer la permutation particulière dans laquelle chaque élément $A[i]$ reçoit la i -ème plus petite priorité. Nous montrons que la probabilité d'apparition de cette permutation vaut exactement $1/n!$. Pour $i = 1, 2, \dots, n$ soit X_i l'événement par lequel l'élément $A[i]$ reçoit la i -ème plus petite priorité. Nous voulons alors calculer la probabilité de voir pour tout i se produire l'événement X_i , probabilité qui est

$$\Pr \{X_1 \cap X_2 \cap X_3 \cap \dots \cap X_{n-1} \cap X_n\} .$$

D'après l'exercice C.2.6 cette probabilité est égale à

$$\Pr \{X_1\} \cdot \Pr \{X_2 \mid X_1\} \cdot \Pr \{X_3 \mid X_2 \cap X_1\} \cdot \Pr \{X_4 \mid X_3 \cap X_2 \cap X_1\} \\ \cdots \Pr \{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr \{X_n \mid X_{n-1} \cap \cdots \cap X_1\} .$$

Nous avons $\Pr \{X_1\} = 1/n$, car c'est la probabilité qu'une priorité prise au hasard dans un ensemble de n éléments soit la plus petite. Ensuite, nous observons que $\Pr \{X_2 \mid X_1\} = 1/(n-1)$; en effet, comme l'élément $A[1]$ a la priorité la plus faible, chacun des $n-1$ éléments restants a une chance égale d'avoir la deuxième priorité la plus faible. Plus généralement, pour $i = 2, 3, \dots, n$, nous avons $\Pr \{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1)$; en effet, vu que les éléments $A[1]$ à $A[i-1]$ ont les $i-1$ priorités les plus faibles (dans l'ordre), chacun des $n-(i-1)$ éléments restants a une chance égale d'avoir la i -ème plus petite priorité. Par conséquent, nous avons

$$\Pr \{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} = \left(\frac{1}{n}\right) \left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right) \left(\frac{1}{1}\right) \\ = \frac{1}{n!} ,$$

et nous avons montré que la probabilité d'obtenir la permutation identité est de $1/n!$.

Nous pouvons généraliser cette démonstration pour qu'elle fonctionne avec n'importe quelle permutation de priorités. Considérons une permutation fixée quelconque $\sigma = \langle \sigma(1), \sigma(2), \dots, \sigma(n) \rangle$ de l'ensemble $\{1, 2, \dots, n\}$. Notons r_i le rang de la priorité assignée à l'élément $A[i]$, sachant que l'élément ayant la j -ème plus petite priorité a le rang j . Si nous définissons X_i comme l'événement par lequel l'élément $A[i]$ reçoit la $\sigma(i)$ -ème plus petite priorité, c'est-à-dire dans lequel $r_i = \sigma(i)$, alors la même démonstration s'applique encore. Si donc nous calculons la probabilité d'obtenir une permutation particulière quelconque, le calcul est identique à celui qui précède, de sorte que la probabilité d'obtenir cette permutation est aussi de $1/n!$. \square

On pourrait penser que, pour démontrer qu'une permutation est une permutation aléatoire uniforme, il suffit de montrer que, pour chaque élément $A[i]$, la probabilité de le voir figurer en position j vaut $1/n$. L'exercice 5.3.4 montre que cette condition moins exigeante est, en fait, insuffisante.

Une meilleure méthode de génération d'une permutation aléatoire consiste à permuter directement le tableau initial. La procédure RANDOMISATION-DIRECTE le fait avec une durée $O(n)$. Dans l'itération i , l'élément $A[i]$ est choisi au hasard parmi les éléments $A[i]$ à $A[n]$. Après l'itération i , il n'y a plus jamais modification de $A[i]$.

RANDOMISATION-DIRECTE(A)

- 1 $n \leftarrow \text{longueur}[A]$
- 2 **pour** $i \leftarrow 1$ à n
- 3 **faire** échanger $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

Nous emploierons un invariant de boucle pour montrer que la procédure RANDOMISATION-DIRECTE produit une permutation aléatoire uniforme. Étant donné un ensemble à n éléments, une k -permutation est une suite contenant k de ces n éléments. (Voir annexe C) Il existe $n!/(n-k)!$ possibilités de telles k -permutations.

Lemme 5.5 *La procédure RANDOMISATION-DIRECTE calcule une permutation aléatoire uniforme.*

Démonstration : Nous utilisons l'invariant de boucle que voici :

Juste avant la i -ème itération de la boucle **pour** des lignes 2–3, pour chaque $(i-1)$ -permutation possible, le sous-tableau $A[1..i-1]$ contient cette $(i-1)$ -permutation avec la probabilité $(n-i+1)!/n!$.

Il faut démontrer que cet invariant est vrai avant la première itération de la boucle, qu'il est conservé par chaque itération et qu'il fournit une propriété utile pour montrer qu'il est vrai quand la boucle se termine.

Initialisation : Considérons la situation juste avant la première itération, donc quand $i = 1$. L'invariant dit que, pour chaque 0-permutation possible, le sous-tableau $A[1..0]$ contient cette 0-permutation avec la probabilité $(n-i+1)!/n! = n!/n! = 1$. Le sous-tableau $A[1..0]$ est vide, et une 0-permutation n'a pas d'éléments. Par conséquent, $A[1..0]$ contient n'importe quelle 0-permutation avec la probabilité 1 ; l'invariant est donc vérifié avant la première itération.

Conservation : Nous supposons que, juste avant la $(i-1)$ -ème itération, chaque $(i-1)$ -permutation possible apparaît dans le sous-tableau $A[1..i-1]$ avec la probabilité $(n-i+1)!/n!$; nous allons montrer que, après la i -ème itération, chaque i -permutation possible apparaît dans le sous-tableau $A[1..i]$ avec la probabilité $(n-i)!/n!$. L'incrément de i pour l'itération suivante conserve alors l'invariant.

Regardons la i -ème itération. Prenons une i -permutation particulière et notons ses éléments par $\langle x_1, x_2, \dots, x_i \rangle$. Cette permutation se compose d'une $(i-1)$ -permutation $\langle x_1, \dots, x_{i-1} \rangle$, suivie de la valeur x_i que l'algorithme place dans $A[i]$. Soit E_1 l'événement par lequel les $i-1$ premières itérations ont créé la $(i-1)$ -permutation particulière $\langle x_1, \dots, x_{i-1} \rangle$ dans $A[1..i-1]$. D'après l'invariant de boucle, $\Pr\{E_1\} = (n-i+1)!/n!$. Soit E_2 l'événement par lequel cette i -ème itération met x_i dans la position $A[i]$. Comme la i -permutation $\langle x_1, \dots, x_i \rangle$ est formée dans $A[1..i]$ uniquement quand se produisent E_1 et E_2 , nous voulons donc calculer $\Pr\{E_2 \cap E_1\}$. D'après l'équation (C.14) nous avons

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} .$$

La probabilité $\Pr\{E_2 \mid E_1\}$ vaut $1/(n-i+1)$; en effet, sur la ligne 3, l'algorithme choisit x_i au hasard parmi les $n-i+1$ valeurs qui occupent les positions $A[i..n]$. Nous avons donc

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} . \end{aligned}$$

Terminaison : À la fin de la boucle, $i = n+1$, et nous avons le résultat que le sous-tableau $A[1..n]$ est une n -permutation donnée avec la probabilité $(n-n)!/n! = 1/n!$.

En conclusion, RANDOMISATION-DIRECTE produit une permutation aléatoire uniforme. \square

Un algorithme randomisé est souvent le moyen le plus simple et le plus efficace de résoudre un problème. Dans ce livre, nous utiliserons de temps à autre des algorithmes randomisés.

Exercices

5.3.1 Le professeur Marceau conteste l'invariant de boucle utilisé dans la démonstration du lemme 5.5. Il met en doute sa validité avant la première itération. Son raisonnement est le suivant : on pourrait, tout aussi bien, déclarer qu'un sous-tableau vide ne contient pas de 0-permutations. Par conséquent, la probabilité qu'un sous-tableau vide contienne une 0-permutation doit être 0, ce qui invalide l'invariant de boucle avant la première itération. Ré-écrire la procédure RANDOMISATION-DIRECTE de façon que l'invariant de boucle associé s'applique à un sous-tableau non vide avant la première itération, et modifier la démonstration du lemme 5.5 pour votre procédure.

5.3.2 Le professeur Kelp décide d'écrire une procédure qui produira aléatoirement une permutation différente en plus de la permutation identité. Il propose la procédure que voici :

```

PERMUTE-SANS-IDENTITÉ(A)
1   $n \leftarrow \text{longueur}[A]$ 
2  pour  $i \leftarrow 1$  à  $n - 1$ 
3      faire échanger  $A[i] \leftrightarrow A[\text{RANDOM}(i + 1, n)]$ 

```

Ce code fait-il ce que le professeur Kelp souhaite ?

5.3.3 Supposez que, au lieu d'échanger l'élément $A[i]$ avec un élément aléatoire du sous-tableau $A[i..n]$, nous l'échangions avec un élément aléatoire pris n'importe où dans le tableau :

```

PERMUTE-AVEC-TOUT(A)
1   $n \leftarrow \text{longueur}[A]$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire échanger  $A[i] \leftrightarrow A[\text{RANDOM}(1, n)]$ 

```

Ce code produit-il une permutation aléatoire uniforme ? Justifiez votre réponse ?

5.3.4 Le professeur Armstrong suggère la procédure suivante pour générer une permutation aléatoire uniforme :

```

PERMUTE-PAR-CYCLE(A)
1   $n \leftarrow \text{longueur}[A]$ 
2   $aux \leftarrow \text{RANDOM}(1, n)$ 
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire  $dec \leftarrow i + aux$ 
5          si  $dec > n$ 
6              alors  $dec \leftarrow dec - n$ 
7               $B[dec] \leftarrow A[i]$ 
8  retourner  $B$ 

```

Montrez que chaque élément $A[i]$ a une probabilité $1/n$ de se retrouver à un certain emplacement, quel qu'il soit, de B . Montrez ensuite que le professeur Armstrong se trompe, en prouvant que la permutation résultante n'est pas uniformément aléatoire.

5.3.5 ★ Prouvez que, dans le tableau P de la procédure PERMUTE-PAR-TRI, la probabilité que tous les éléments soient uniques est au moins $1 - 1/n$.

5.3.6 Expliquez comment implémenter l'algorithme PERMUTE-PAR-TRI pour traiter le cas où deux ou plusieurs priorités sont identiques. En d'autres termes, votre algorithme doit produire une permutation aléatoire uniforme même si deux ou plusieurs priorités sont identiques.

5.4 ★ ANALYSE PROBABILISTE ET AUTRES EMPLOIS DES VARIABLES INDICATRICES

Cette section avancée montre d'autres utilisations de l'analyse probabiliste, et ce sur quatre exemples. Le premier détermine la probabilité que, dans une pièce contenant k personnes, il se trouve deux personnes qui aient la même date de naissance. Le deuxième exemple étudie le lancer aléatoire de ballons en direction de paniers. Le troisième étudie les « suites » de piles successifs dans les lancers d'une pièce de monnaie. Le dernier exemple étudie une variante du problème de l'embauche, dans laquelle vous devez prendre des décisions sans interviewer réellement toutes les candidates.

5.4.1 Le paradoxe des anniversaires

Notre premier exemple est le *paradoxe des anniversaires*. Combien doit-il y avoir de gens dans une pièce pour qu'il y ait 50% de chances que deux personnes soient nées le même jour de l'année? Chose surprenante, la réponse est qu'il suffit d'un petit nombre de personnes. Le paradoxe est que ce nombre est, en fait, très inférieur au nombre de jours dans une année, voire à la moitié du nombre de jours dans une année, comme nous allons le voir.

Pour répondre à cette question, nous indexons les personnes présentes dans la pièce à l'aide des entiers $1, 2, \dots, k$, où k est le nombre total de ces personnes. Nous ignorerons la problématique des années bissextiles et partirons du principe que toutes les années ont $n = 365$ jours. Pour $i = 1, 2, \dots, k$, soit b_i le jour anniversaire de la personne i , sachant que $1 \leq b_i \leq n$. Nous supposerons également que les anniversaires sont distribués de manière uniforme sur les n jours de l'année, de sorte que $\Pr\{b_i = r\} = 1/n$ pour $i = 1, 2, \dots, k$ et $r = 1, 2, \dots, n$.

La probabilité que deux personnes données, par exemple i et j , aient le même anniversaire dépend de l'indépendance éventuelle de la sélection aléatoire des anniversaires. Nous supposerons dorénavant que les anniversaires sont indépendants, de

sorte que la probabilité que l'anniversaire de i et celui de j tombent le jour r est

$$\begin{aligned} \Pr \{b_i = r \text{ and } b_j = r\} &= \Pr \{b_i = r\} \Pr \{b_j = r\} \\ &= 1/n^2 . \end{aligned}$$

Par conséquent, la probabilité que les deux anniversaires tombent le même jour est

$$\begin{aligned} \Pr \{b_i = b_j\} &= \sum_{r=1}^n \Pr \{b_i = r \text{ et } b_j = r\} \\ &= \sum_{r=1}^n (1/n^2) \\ &= 1/n . \end{aligned} \tag{5.7}$$

De manière plus intuitive, une fois choisi b_i , la probabilité que b_j soit choisi de façon à être le même jour de l'année est égale à $1/n$. Ainsi, la probabilité que i et j aient le même anniversaire est égale à la probabilité que l'anniversaire de l'une de ces personnes tombe un certain jour de l'année. Notez, cependant, que cette coïncidence dépend de l'hypothèse selon laquelle les anniversaires sont indépendants.

Pour déterminer la probabilité d'avoir au moins 2 personnes sur k qui aient le même anniversaire, nous pouvons regarder l'événement complémentaire. La probabilité qu'il y ait au moins deux anniversaires identiques est égale à 1 moins la probabilité que tous les anniversaires soient différents. L'évènement dans lequel k personnes ont des anniversaires distincts est

$$B_k = \bigcap_{i=1}^k A_i ,$$

où A_i est l'évènement par lequel l'anniversaire de la personne i est différent de celui de la personne j pour tout $j < i$. Comme nous pouvons écrire $B_k = A_k \cap B_{k-1}$, nous obtenons d'après l'équation (C.16) la récurrence

$$\Pr \{B_k\} = \Pr \{B_{k-1}\} \Pr \{A_k \mid B_{k-1}\} , \tag{5.8}$$

où nous prenons $\Pr \{B_1\} = \Pr \{A_1\} = 1$ comme condition initiale. En d'autres termes, la probabilité que b_1, b_2, \dots, b_k soient des anniversaires distincts est égale à la probabilité que b_1, b_2, \dots, b_{k-1} soient des anniversaires distincts multipliée par la probabilité que $b_k \neq b_i$ pour $i = 1, 2, \dots, k-1$, sachant que b_1, b_2, \dots, b_{k-1} sont distincts. Si b_1, b_2, \dots, b_{k-1} sont distincts, la probabilité conditionnelle que $b_k \neq b_i$ pour $i = 1, 2, \dots, k-1$ est égale à $\Pr \{A_k \mid B_{k-1}\} = (n - k + 1)/n$, vu que, sur les n jours, il y en a $n - (k - 1)$ qui ne sont pas pris. Nous appliquons de manière réitérée

la récurrence (5.8) pour obtenir

$$\begin{aligned}
 \Pr \{B_k\} &= \Pr \{B_{k-1}\} \Pr \{A_k \mid B_{k-1}\} \\
 &= \Pr \{B_{k-2}\} \Pr \{A_{k-1} \mid B_{k-2}\} \Pr \{A_k \mid B_{k-1}\} \\
 &\vdots \\
 &= \Pr \{B_1\} \Pr \{A_2 \mid B_1\} \Pr \{A_3 \mid B_2\} \cdots \Pr \{A_k \mid B_{k-1}\} \\
 &= 1 \cdot \left(\frac{n-1}{n}\right) \left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\
 &= 1 \cdot \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right).
 \end{aligned}$$

L'inégalité (3.11), $1+x \leq e^x$, nous donne

$$\begin{aligned}
 \Pr \{B_k\} &\leq e^{-1/n} e^{-2/n} \cdots e^{-(k-1)/n} \\
 &= e^{-\sum_{i=1}^{k-1} i/n} \\
 &= e^{-k(k-1)/2n} \\
 &\leq 1/2
 \end{aligned}$$

quand $-k(k-1)/2n \leq \ln(1/2)$. La probabilité que tous les k anniversaires soient distincts est au plus $1/2$ quand $k(k-1) \geq 2n \ln 2$ ou, en résolvant l'équation quadratique, quand $k \geq (1 + \sqrt{1 + (8 \ln 2)n})/2$. Pour $n = 365$, nous devons avoir $k \geq 23$. Par conséquent, s'il y a au moins 23 personnes dans une pièce, la probabilité qu'il y en ait au moins deux qui aient le même anniversaire est d'au moins $1/2$. Sur Mars où l'année fait 669 jours, il faut avoir 31 Martiens pour obtenir le même effet.

a) Analyse via variables indicatrices

Nous pouvons employer des variables indicatrices pour fournir une analyse plus simple, mais approximative, du paradoxe des anniversaires. Pour chaque paire (i, j) formée parmi les k personnes, nous définissons la variable indicatrice X_{ij} , pour $1 \leq i < j \leq k$, de la façon suivante

$$\begin{aligned}
 X_{ij} &= I \{\text{personne } i \text{ et personne } j \text{ ont le même anniversaire}\} \\
 &= \begin{cases} 1 & \text{si personne } i \text{ et personne } j \text{ ont le même anniversaire,} \\ 0 & \text{sinon.} \end{cases}
 \end{aligned}$$

D'après l'équation (5.7), la probabilité que deux personnes aient le même anniversaire est de $1/n$, et donc, d'après le lemme 5.1, nous avons

$$\begin{aligned}
 E[X_{ij}] &= \Pr \{\text{personne } i \text{ et personne } j \text{ ont le même anniversaire}\} \\
 &= 1/n.
 \end{aligned}$$

Si X est la variable aléatoire qui compte le nombre de paires d'individus ayant le même anniversaire, nous avons

$$X = \sum_{i=1}^k \sum_{j=i+1}^k X_{ij} .$$

En prenant les espérances des deux côtés et en appliquant la linéarité de l'espérance, nous obtenons

$$\begin{aligned} E[X] &= E \left[\sum_{i=1}^k \sum_{j=i+1}^k X_{ij} \right] \\ &= \sum_{i=1}^k \sum_{j=i+1}^k E[X_{ij}] \\ &= \binom{k}{2} \frac{1}{n} \\ &= \frac{k(k-1)}{2n} . \end{aligned}$$

Ainsi, quand $k(k-1) \geq 2n$, le nombre attendu de paires de personnes ayant le même anniversaire est au moins 1. Si donc nous avons au moins $\sqrt{2n} + 1$ individus dans une pièce, nous pouvons espérer que deux personnes au moins auront le même anniversaire. Pour $n = 365$, si $k = 28$, le nombre attendu de paires ayant le même anniversaire est $(28 \cdot 27) / (2 \cdot 365) \approx 1,0356$. Par conséquent, avec au moins 28 personnes, on peut s'attendre à trouver au moins une paire de personnes ayant le même anniversaire. Sur Mars où l'année fait 669 jours, il faut au moins 38 Martiens. La première analyse, basée sur les seules probabilités, calculait le nombre de personnes requis pour que la probabilité qu'il y ait une paire d'anniversaires identiques dépasse $1/2$; la seconde analyse, qui emploie des variables indicatrices, calcule le nombre de personnes requis pour que le nombre attendu d'anniversaires identiques soit égal à 1. Le nombre exact de personnes n'est pas le même dans les deux situations, mais les valeurs sont les mêmes asymptotiquement : $\Theta(\sqrt{n})$.

5.4.2 Ballons et paniers

Considérons le lancer aléatoire de ballons identiques vers b paniers numérotés $1, 2, \dots, b$. Les lancers sont indépendants, et pour chaque lancer le ballon a la même probabilité d'arriver dans l'un quelconque des paniers. La probabilité qu'un ballon atterrisse dans un quelconque panier donné est $1/b$. Le lancer des ballons est donc une suite d'épreuves de Bernoulli (voir Annexe C.4) avec une probabilité $1/b$ de succès, succès signifiant ici que le ballon arrive dans le panier donné. Ce modèle est particulièrement utile pour l'analyse du hachage (voir Chapitre 11), et nous pouvons répondre à une foule de questions intéressantes concernant le processus du lancer de

ballons. (Le problème C.1 posera des questions supplémentaires sur les ballons et paniers.)

Combien de ballons arrivent dans un panier donné ? Le nombre de ballons qui tombent dans un panier donné obéit à la distribution binomiale $b(k; n, 1/b)$. Si on lance n ballons, l'équation (C.36) dit que le nombre attendu de ballons qui tombent dans le panier donné est n/b .

Combien faut-il lancer de ballons, en moyenne, pour qu'un panier donné contienne un ballon ? Le nombre de lancers à effectuer pour que le panier donné reçoive un ballon obéit à la distribution géométrique avec une probabilité $1/b$ et, d'après l'équation (C.31), le nombre attendu de lancers est $1/(1/b) = b$.

Combien faut-il lancer de ballons pour que chaque panier contienne au moins un ballon ? Appelons « coup au but » un lancer dans lequel un ballon tombe dans un panier vide. Nous voulons connaître le nombre attendu de lancers n exigé pour avoir b coups au but.

Les coups au but peuvent servir à diviser les n lancers en salves. La i -ème salve regroupe les lancers qui suivent le $(i - 1)$ -ème coup au but jusqu'au i -ème coup au but. La première salve se compose du premier lancer, puisque nous sommes certains d'avoir un coup au but quand tous les paniers sont vides. Pour chaque lancer de la i -ème salve, il y a $i - 1$ paniers qui contiennent des ballons et $b - i + 1$ paniers vides. Par conséquent, pour chaque lancer de la i -ème salve, la probabilité d'obtenir un coup au but est $(b - i + 1)/b$.

Soit n_i le nombre de lancers de la i -ème salve. Il s'ensuit que le nombre de lancers requis pour avoir b coups au but est $n = \sum_{i=1}^b n_i$. Chaque variable aléatoire n_i a une distribution géométrique avec la probabilité de succès $(b - i + 1)/b$ et, d'après l'équation (C.31),

$$E[n_i] = \frac{b}{b - i + 1}.$$

En raison de la linéarité de l'espérance,

$$\begin{aligned} E[n] &= E\left[\sum_{i=1}^b n_i\right] = \sum_{i=1}^b E[n_i] = \sum_{i=1}^b \frac{b}{b - i + 1} = b \sum_{i=1}^b \frac{1}{i} \\ &= b(\ln b + O(1)). \end{aligned}$$

La dernière ligne découle de la borne (A.7) sur la série harmonique. Il faut donc environ $b \ln b$ lancers pour que nous puissions espérer que chaque panier contienne un ballon. Ce problème porte aussi le nom de **problème du collecteur de coupons** ; il dit qu'une personne qui essaie de recueillir chacun de b coupons différents doit se procurer environ $b \ln b$ coupons obtenus au hasard pour pouvoir réussir.

5.4.3 Suites

Supposez que vous jetez n fois en l'air une pièce non faussée. Quelle est la plus longue suite de piles consécutifs que vous vous espérez voir ? La réponse est $\Theta(\lg n)$, comme va le montrer l'étude qui suit.

Nous commencerons par prouver que la longueur attendue de la plus longue suite de piles est $O(\lg n)$. La probabilité que chaque lancer de la pièce donne pile est $1/2$. Soit A_{ik} l'événement par lequel une suite de piles de longueur au moins k commence au i -ème lancer ou, plus précisément, l'événement par lequel les k lancers consécutifs de la pièce $i, i+1, \dots, i+k-1$ donnent uniquement des piles, avec $1 \leq k \leq n$ et $1 \leq i \leq n-k+1$. Comme les lancers sont mutuellement indépendants, pour un événement donné A_{ik} , la probabilité que tous les k lancers produisent des piles est

$$\Pr \{A_{ik}\} = 1/2^k . \quad (5.9)$$

Pour $k = 2 \lceil \lg n \rceil$,

$$\begin{aligned} \Pr \{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\ &\leq 1/2^{2\lg n} \\ &= 1/n^2 , \end{aligned}$$

et donc la probabilité qu'une suite de piles de longueur au moins égale à $2 \lceil \lg n \rceil$ commence à la position i est assez faible. Il existe au plus $n - 2 \lceil \lg n \rceil + 1$ positions où une telle suite peut commencer. La probabilité qu'une suite de piles de longueur au moins égale à $2 \lceil \lg n \rceil$ commence à un emplacement quelconque est donc

$$\begin{aligned} \Pr \left\{ \bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil} \right\} &\leq \sum_{i=1}^{n-2\lceil \lg n \rceil+1} 1/n^2 \\ &< \sum_{i=1}^n 1/n^2 \\ &= 1/n , \end{aligned} \quad (5.10)$$

puisque, d'après l'inégalité de Boole (C.18), la probabilité d'une union d'événements est au plus égale à la somme des probabilités des événements individuels. (Notez que l'inégalité de Boole reste vraie pour des événements tels que ceux-là qui ne sont pas indépendants.)

Nous employons maintenant l'inégalité (5.10) pour borner la longueur de la plus longue suite. Pour $j = 0, 1, 2, \dots, n$, soit L_j l'événement par lequel la plus longue suite de piles a une longueur d'exactly j , et soit L la longueur de la plus longue suite. D'après la définition de l'espérance,

$$E[L] = \sum_{j=0}^n j \Pr \{L_j\} . \quad (5.11)$$

Nous pourrions essayer d'évaluer cette somme en utilisant des bornes supérieures sur chaque $\Pr \{L_j\}$ semblables à celles calculées dans l'inégalité (5.10). Hélas, cette méthode donnerait des bornes faibles. Toutefois, nous pouvons utiliser une certaine intuition, obtenue via l'analyse qui précède, pour avoir une borne correcte. De manière informelle, nous observons que, pour aucun des divers termes de la sommation dans l'équation (5.11), les facteurs j et $\Pr \{L_j\}$ ne sont tous les deux grands en même temps. Pourquoi ? Quand $j \geq 2 \lceil \lg n \rceil$, $\Pr \{L_j\}$ est très petit ; et quand $j < 2 \lceil \lg n \rceil$, j est assez petit. De manière plus formelle, nous remarquons que les événements L_j pour $j = 0, 1, \dots, n$ sont disjoints, et donc que la probabilité qu'une suite de piles de longueur au moins $2 \lceil \lg n \rceil$ commence à un endroit quelconque est $\sum_{j=2 \lceil \lg n \rceil}^n \Pr \{L_j\}$. D'après l'inégalité (5.10), nous avons $\sum_{j=2 \lceil \lg n \rceil}^n \Pr \{L_j\} < 1/n$. En outre, en remarquant que $\sum_{j=0}^n \Pr \{L_j\} = 1$, nous avons $\sum_{j=0}^{2 \lceil \lg n \rceil - 1} \Pr \{L_j\} \leq 1$. Par conséquent, nous obtenons

$$\begin{aligned}
 E[L] &= \sum_{j=0}^n j \Pr \{L_j\} \\
 &= \sum_{j=0}^{2 \lceil \lg n \rceil - 1} j \Pr \{L_j\} + \sum_{j=2 \lceil \lg n \rceil}^n j \Pr \{L_j\} \\
 &< \sum_{j=0}^{2 \lceil \lg n \rceil - 1} (2 \lceil \lg n \rceil) \Pr \{L_j\} + \sum_{j=2 \lceil \lg n \rceil}^n n \Pr \{L_j\} \\
 &= 2 \lceil \lg n \rceil \sum_{j=0}^{2 \lceil \lg n \rceil - 1} \Pr \{L_j\} + n \sum_{j=2 \lceil \lg n \rceil}^n \Pr \{L_j\} \\
 &< 2 \lceil \lg n \rceil \cdot 1 + n \cdot (1/n) \\
 &= O(\lg n) .
 \end{aligned}$$

Les chances qu'il y ait plus de $r \lceil \lg n \rceil$ piles d'affilée diminuent rapidement avec r . Pour $r \geq 1$, la probabilité qu'une suite de $r \lceil \lg n \rceil$ piles commence à la position i est

$$\begin{aligned}
 \Pr \{A_{i, r \lceil \lg n \rceil}\} &= 1/2^{r \lceil \lg n \rceil} \\
 &\leq 1/n^r .
 \end{aligned}$$

Ainsi, la probabilité est au plus de $n/n^r = 1/n^{r-1}$ que la plus longue suite fasse au moins $r \lceil \lg n \rceil$, ou dit autrement, la probabilité est au moins de $1 - 1/n^{r-1}$ que la plus longue suite ait une longueur inférieure à $r \lceil \lg n \rceil$.

À titre d'exemple, pour $n = 1000$ lancers de pièce, la probabilité d'avoir au moins $2 \lceil \lg n \rceil = 20$ piles d'affilée est au plus $1/n = 1/1000$. Les chances d'avoir une suite de piles plus longue que $3 \lceil \lg n \rceil = 30$ sont au plus $1/n^2 = 1/1\,000\,000$.

Nous allons maintenant prouver l'existence d'une borne inférieure complémentaire : la longueur attendue de la plus longue suite de piles dans n lancers de pièce

est $\Omega(\lg n)$. Pour démontrer la validité de cette borne, nous cherchons des suites de longueur s en partitionnant les n lancers en environ n/s groupes de s lancers chacun. Si nous choisissons $s = \lfloor (\lg n)/2 \rfloor$, nous pouvons montrer qu'il est probable qu'un de ces groupes au moins ne contienne que des piles, et donc qu'il est probable que la plus longue suite ait une longueur d'au moins $s = \Omega(\lg n)$. Nous montrerons ensuite que la plus longue suite a une longueur attendue de $\Omega(\lg n)$.

Nous partitionnons les n lancers en au moins $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ groupes de $\lfloor (\lg n)/2 \rfloor$ lancers consécutifs, et nous bornons la probabilité qu'il n'y ait aucun groupe ne contenant que des piles. D'après l'équation (5.9), la probabilité que le groupe commençant à la position i ne contienne que des piles est

$$\begin{aligned} \Pr \{A_{i, \lfloor (\lg n)/2 \rfloor}\} &= 1/2^{\lfloor (\lg n)/2 \rfloor} \\ &\geq 1/\sqrt{n}. \end{aligned}$$

La probabilité qu'une suite de piles de longueur au moins $\lfloor (\lg n)/2 \rfloor$ ne commence pas à la position i est donc au plus de $1 - 1/\sqrt{n}$. Comme les $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ groupes sont formés de lancers indépendants mutuellement exclusifs, la probabilité que chacun de ces groupes *ne soit pas* une suite de longueur $\lfloor (\lg n)/2 \rfloor$ est au plus

$$\begin{aligned} (1 - 1/\sqrt{n})^{\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor} &\leq (1 - 1/\sqrt{n})^{n/\lfloor (\lg n)/2 \rfloor - 1} \\ &\leq (1 - 1/\sqrt{n})^{2n/\lg n - 1} \\ &\leq e^{-(2n/\lg n - 1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n). \end{aligned}$$

Pour cette discussion, nous nous sommes servis de l'inégalité (3.11), $1 + x \leq e^x$, et du fait (que vous pourrez vérifier) que $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ pour n suffisamment grand.

Par conséquent, la probabilité que la plus longue suite ait une longueur supérieure à $\lfloor (\lg n)/2 \rfloor$ est

$$\sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^n \Pr \{L_j\} \geq 1 - O(1/n). \quad (5.12)$$

Nous pouvons maintenant calculer une borne inférieure pour la longueur attendue de la plus longue suite, en partant de l'équation (5.11) et en raisonnant d'une manière

semblable à notre analyse de la borne supérieure :

$$\begin{aligned}
 E[L] &= \sum_{j=0}^n j \Pr\{L_j\} \\
 &= \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} j \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^n j \Pr\{L_j\} \\
 &\geq \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^n \lfloor (\lg n)/2 \rfloor \Pr\{L_j\} \\
 &= 0 \cdot \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} \Pr\{L_j\} + \lfloor (\lg n)/2 \rfloor \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^n \Pr\{L_j\} \\
 &\geq 0 + \lfloor (\lg n)/2 \rfloor (1 - O(1/n)) \quad (\text{d'après l'inégalité (5.12)}) \\
 &= \Omega(\lg n) .
 \end{aligned}$$

Comme pour le paradoxe des anniversaires, nous pouvons obtenir une analyse plus simple, mais approximative, en utilisant des variables indicatrices. Soit $X_{ik} = I\{A_{ik}\}$ la variable indicatrice associée à une suite de piles de longueur au moins k commençant au i -ème lancer. Pour compter le nombre total de telles suites, nous définissons

$$X = \sum_{i=1}^{n-k+1} X_{ik} .$$

En prenant les espérances et en appliquant la linéarité de l'espérance, nous avons

$$\begin{aligned}
 E[X] &= E\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
 &= \sum_{i=1}^{n-k+1} E[X_{ik}] \\
 &= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
 &= \sum_{i=1}^{n-k+1} 1/2^k \\
 &= \frac{n-k+1}{2^k} .
 \end{aligned}$$

En appliquant ce résultat à différentes valeurs de k , nous pouvons calculer le nombre attendu de suites de longueur k . Si ce nombre est grand (très supérieur à 1), alors on peut s'attendre à voir moult suites de longueur k et la probabilité d'ap-

partition d'une telle suite est élevée. Si ce nombre est faible (très inférieur à 1), alors on peut s'attendre à ne voir que très peu de suites de longueur k et la probabilité d'apparition d'une telle suite est réduite. Si $k = c \lg n$, c étant une certaine constante positive, on obtient

$$\begin{aligned} E[X] &= \frac{n - c \lg n + 1}{2^{c \lg n}} \\ &= \frac{n - c \lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}). \end{aligned}$$

Si c est grand, le nombre attendu de suites de longueur $c \lg n$ est très faible, et l'on en conclut qu'il y a peu de chances de voir apparaître une telle suite. En revanche, si $c < 1/2$, alors on obtient $E[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, et l'on peut s'attendre à voir un grand nombre de suites de longueur $(1/2) \lg n$. Par conséquent, la probabilité d'apparition d'une suite ayant cette longueur est très forte. À partir de ces seules estimations, l'on peut déduire que la longueur attendue de la plus longue suite est $\Theta(\lg n)$.

5.4.4 Le problème de l'embauche en ligne

Comme dernier exemple, nous allons considérer une variante du problème de l'embauche. On suppose désormais que l'on ne veut pas interviewer toutes les candidates pour trouver la meilleure. On ne veut pas, non plus, passer son temps à embaucher et congédier des candidates. On veut plutôt sélectionner une candidate proche de l'optimum, afin de ne procéder qu'à une seule embauche. L'entreprise impose la contrainte suivante : après chaque entrevue, il faut soit proposer immédiatement le poste à la candidate, soit lui dire qu'elle ne fait pas l'affaire. Quel est l'équilibre entre la minimisation du nombre d'entrevues et la maximisation de la qualité de la candidate engagée ?

Nous pouvons modéliser le problème de la manière suivante. Après avoir vu une postulante, nous lui donnons une note ; soit $score(i)$ la note de la i -ème candidate, en supposant que deux postulantes quelconques n'obtiennent jamais la même note. Lorsque nous aurons vu j candidates, nous saurons laquelle des j a la meilleure note, mais nous ne saurons pas si l'une des $n - j$ candidates restantes serait susceptible d'avoir une note encore meilleure. Nous décidons d'adopter la stratégie suivante : nous sélectionnons un entier positif $k < n$, nous interviewons puis refusons les k premières postulantes, et enfin nous engageons la première postulante suivante ayant une note supérieure à toutes les candidates qui l'ont précédée. S'il s'avère que la candidate la plus qualifiée figurait parmi les k premières interviewées, alors nous embauchons la n -ème postulante. Cette stratégie est formalisée dans la procédure

MAXIMUM-EN-LIGNE(k, n), donnée ci-après. La procédure MAXIMUM-EN-LIGNE retourne l'indice de la candidate que nous voulons engager.

```

MAXIMUM-EN-LIGNE( $k, n$ )
1  meilleur score  $\leftarrow -\infty$ 
2  pour  $i \leftarrow 1$  à  $k$ 
3      faire si  $score(i) > meilleur\ score$ 
4          alors meilleur score  $\leftarrow score(i)$ 
5  pour  $i \leftarrow k + 1$  à  $n$ 
6      faire si  $score(i) > meilleur\ score$ 
7          alors retourner  $i$ 
8  retourner  $n$ 

```

Nous voulons déterminer, pour chaque valeur possible de k , la probabilité d'engager la candidate la plus qualifiée. Nous choisirons ensuite la valeur optimale pour k et appliquerons notre stratégie avec cette valeur. Pour l'instant, supposons k fixé. Soit $M(j) = \max_{1 \leq i \leq j} \{score(i)\}$ la note maximale parmi les postulantes 1 à j . Soit S l'événement par lequel nous parvenons à sélectionner la candidate la plus qualifiée, et soit S_i l'événement par lequel la candidate optimale est la i -ème interviewée. Comme les différents S_i sont disjoints, nous avons $\Pr\{S\} = \sum_{i=1}^n \Pr\{S_i\}$. En remarquant que nous ne réussissons jamais quand la candidate la plus qualifiée figure parmi les k premières, nous avons $\Pr\{S_i\} = 0$ pour $i = 1, 2, \dots, k$. Par conséquent, nous obtenons

$$\Pr\{S\} = \sum_{i=k+1}^n \Pr\{S_i\}$$

Calculons maintenant $\Pr\{S_i\}$. Pour que nous réussissions quand la candidate optimale est la i -ème, il faut que se produisent deux choses. Primo, la candidate la plus qualifiée doit être en position i , événement que nous noterons B_i . Secundo, l'algorithme ne doit sélectionner aucune des postulantes $k + 1$ à $i - 1$, ce qui ne se produit que si, pour chaque j tel que $k + 1 \leq j \leq i - 1$, nous trouvons que $score(j) < meilleur\ score$ sur la ligne 6. (Comme les notes sont uniques, nous ignorons la possibilité d'avoir $score(j) = meilleur\ score$.) En d'autres termes, il faut que toutes les valeurs $score(k + 1)$ à $score(i - 1)$ soient inférieures à $M(k)$; s'il y en a qui sont plus grandes que $M(k)$, à la place nous retournons l'indice de la première valeur supérieure. Prenons O_i pour noter l'événement par lequel aucune des postulantes $k + 1$ à $i - 1$ n'est choisie. Fort heureusement, les deux événements B_i et O_i sont indépendants. L'événement O_i dépend uniquement de l'ordre relatif des valeurs des positions 1 à $i - 1$, alors que B_i dépend uniquement de ce que la valeur en position i est ou non supérieure aux valeurs de toutes les autres positions. L'ordre des valeurs des positions 1 à $i - 1$ n'affecte pas le fait que la valeur en position i est ou non plus grande que toutes ces valeurs, et la valeur en position i n'affecte pas l'ordre des valeurs des positions 1 à $i - 1$. Nous pouvons donc appliquer l'équation (C.15) pour obtenir

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\} \Pr\{O_i\} .$$

La probabilité $\Pr\{B_i\}$ est visiblement $1/n$, car le maximum a une chance égale d'occuper l'une quelconque des n positions. Pour que se produise l'événement O_i , la valeur maximale des positions 1 à $i - 1$ doit être l'une des k premières positions, et elle a une probabilité égale d'être l'une quelconque de ces $i - 1$ positions. Par conséquent, $\Pr\{O_i\} = k/(i - 1)$ et $\Pr\{S_i\} = k/(n(i - 1))$. D'après l'équation (5.4.4), nous avons

$$\Pr\{S\} = \sum_{i=k+1}^n \Pr\{S_i\} = \sum_{i=k+1}^n \frac{k}{n(i-1)} = \frac{k}{n} \sum_{i=k+1}^n \frac{1}{i-1} = \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}.$$

Nous approximons par des intégrales pour encadrer cette sommation. D'après les inégalités (A.12), nous avons

$$\int_k^n \frac{1}{x} dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x} dx.$$

L'évaluation de ces intégrales définies nous donne les bornes

$$\frac{k}{n}(\ln n - \ln k) \leq \Pr\{S\} \leq \frac{k}{n}(\ln(n-1) - \ln(k-1)),$$

qui fournissent un encadrement assez serré pour $\Pr\{S\}$. Comme nous souhaitons maximiser notre probabilité de succès, concentrons-nous sur le choix de la valeur de k susceptible de maximiser la borne inférieure de $\Pr\{S\}$. (D'ailleurs, il est plus facile de maximiser l'expression de la borne inférieure que l'expression de la borne supérieure.) En dérivant l'expression $(k/n)(\ln n - \ln k)$ par rapport à k , nous obtenons

$$\frac{1}{n}(\ln n - \ln k - 1).$$

En rendant cette dérivée égale à 0, nous voyons que la borne inférieure de la probabilité est maximisée quand $\ln k = \ln n - 1 = \ln(n/e)$ ou, de manière équivalente, quand $k = n/e$. Si donc nous mettons en œuvre notre stratégie avec $k = n/e$, nous réussirons à embaucher notre candidate la plus qualifiée avec une probabilité d'au moins $1/e$.

Exercices

5.4.1 Combien doit-il y avoir de personnes dans une pièce pour que la probabilité que quelqu'un ait le même anniversaire que vous soit au moins de $1/2$? Combien doit-il y avoir de personnes pour que la probabilité que deux personnes au moins aient leur anniversaire le 14 juillet soit supérieure à $1/2$?

5.4.2 On jette des ballons dans b paniers. Chaque lancer est indépendant, et chaque ballon a une chance égale de finir dans n'importe quel panier. Quel est le nombre attendu de lancers pour qu'au moins l'un des paniers contienne deux ballons?

5.4.3 * Pour l'analyse du paradoxe des anniversaires, est-il important que les anniversaires soient mutuellement indépendants, ou l'indépendance au niveau de la paire suffit-elle ? Justifiez votre réponse.

5.4.4 * Combien de personnes faut-il inviter à une soirée pour que l'on ait des chances d'avoir *trois* personnes ayant le même anniversaire ?

5.4.5 * Quelle est la probabilité qu'une k -chaîne sur un ensemble de taille n soit en fait une k -permutation ? En quoi cette question se rattache-t-elle au paradoxe des anniversaires ?

5.4.6 * On lance n ballons dans n paniers, chaque lancer étant indépendant et le ballon ayant une chance égale d'atterrir dans n'importe quel panier. Quel est le nombre attendu de paniers vides ? Quel est le nombre attendu de paniers contenant exactement un ballon ?

5.4.7 * Affinez la borne inférieure de la longueur de la suite en montrant que, dans n lancers d'une pièce non faussée, la probabilité qu'il n'y ait aucune suite de piles consécutifs plus longue que $\lg n - 2 \lg \lg n$ est inférieure à $1/n$.

PROBLÈMES

5.1. Comptage probabiliste

Un compteur de b bits permet, en principe, de compter jusqu'à $2^b - 1$. Le **comptage probabiliste** de R. Morris permet d'aller beaucoup plus loin, au prix d'une certaine perte de précision.

Une valeur de compteur i représente un comptage de n_i pour $i = 0, 1, \dots, 2^b - 1$, les n_i formant une suite croissante de valeurs non négatives. On suppose que la valeur initiale du compteur est 0, représentant un comptage de $n_0 = 0$. L'opération INCREMENT opère sur un compteur contenant la valeur i d'une manière probabiliste. Si $i = 2^b - 1$, alors il y a erreur de dépassement de capacité. Autrement, le compteur soit augmente de 1 avec la probabilité $1/(n_{i+1} - n_i)$, soit reste inchangé avec la probabilité $1 - 1/(n_{i+1} - n_i)$.

Si l'on prend $n_i = i$ pour tout $i \geq 0$, alors le compteur est un compteur classique. Les choses deviennent plus intéressantes si l'on prend, disons, $n_i = 2^{i-1}$ pour $i > 0$ ou $n_i = F_i$ (i -ème nombre de Fibonacci—voir Section 3.2).

Pour ce problème, on suppose que n_{2^b-1} est suffisamment grand pour que la probabilité d'une erreur de dépassement de capacité soit négligeable.

- Montrez que la valeur attendue représentée par le compteur après n opérations INCREMENT est exactement n .
- L'analyse de la variance du comptage représenté par le compteur dépend de la suite des n_i . Prenons un cas simple : $n_i = 100i$ pour tout $i \geq 0$. Donnez une estimation de la variance de la valeur représentée par le registre après n opérations INCREMENT.

5.2. Recherche dans un tableau non trié

Ce problème étudie trois algorithmes relatifs à la recherche d'une valeur x dans un tableau non trié A à n éléments.

Considérons la stratégie randomisée suivante : on prend au hasard un indice i pour A . Si $A[i] = x$, alors le traitement se termine ; sinon, on continue la recherche en prenant au hasard un nouvel indice pour A . On continue à prendre au hasard des indices pour A jusqu'à ce que l'on trouve un indice j tel que $A[j] = x$, ou jusqu'à ce que l'on ait testé chaque élément de A . Notez ceci : comme nous piochons chaque fois dans tout l'ensemble des indices, nous risquons d'examiner un certain élément plus d'une fois.

- Écrivez le pseudo code d'une procédure RECHERCHE-ALÉATOIRE qui mette en œuvre la stratégie précédemment exposée. Vérifiez que votre algorithme se termine quand on a pioché tous les indices de A .
- On suppose qu'il y a exactement un seul indice i tel que $A[i] = x$. Quel est le nombre attendu d'indices de A qu'il faut piocher pour que la recherche de x aboutisse et que la procédure RECHERCHE-ALÉATOIRE prenne fin ?
- Afin de généraliser la solution de la partie (b), on suppose qu'il y a $k \geq 1$ indices i tels que $A[i] = x$. Quel est le nombre attendu d'indices de A qu'il faut piocher pour que la recherche de x aboutisse et que la procédure RECHERCHE-ALÉATOIRE prenne fin ? La réponse doit être une fonction de n et k .
- On suppose qu'il n'existe aucun indice i tel que $A[i] = x$. Quel est le nombre attendu d'indices de A qu'il faut piocher pour que tous les éléments de A aient été testés et que la procédure RECHERCHE-ALÉATOIRE prenne fin ?

Considérons maintenant un algorithme de recherche linéaire déterministe, que nous appellerons RECHERCHE-DÉTERMINISTE. L'algorithme parcourt A dans l'ordre pour trouver x , testant successivement $A[1], A[2], A[3], \dots, A[n]$ jusqu'à ce que soit vérifié $A[i] = x$ ou que soit atteinte la fin du tableau. On admet l'équiprobabilité de toutes les permutations potentielles du tableau donné en entrée.

- On suppose qu'il y a exactement un seul indice i tel que $A[i] = x$. Quel est le temps d'exécution attendu pour RECHERCHE-DÉTERMINISTE ? Quel est le temps d'exécution de RECHERCHE-DÉTERMINISTE dans le cas le plus défavorable ?
- Afin de généraliser la solution de la partie (e), on suppose qu'il existe $k \geq 1$ indices i tels que $A[i] = x$. Quel est le temps d'exécution attendu pour RECHERCHE-DÉTERMINISTE ? Quel est le temps d'exécution de RECHERCHE-DÉTERMINISTE dans le cas le plus défavorable ? La réponse doit être une fonction de n et k .
- On suppose qu'il n'existe aucun indice i tel que $A[i] = x$. Quel est le temps d'exécution attendu pour RECHERCHE-DÉTERMINISTE ? Quel est le temps d'exécution de RECHERCHE-DÉTERMINISTE dans le cas le plus défavorable ?

Considérons enfin un algorithme randomisé RECHERCHE-MÉLANGE qui commence par permuter de manière aléatoire le tableau initial, puis qui lance la recherche linéaire déterministe, précédemment exposée, sur le tableau permuté résultant.

- h.* k désignant le nombre d'indices i tels que $A[i] = x$, donnez le temps d'exécution attendu et le temps d'exécution du cas le plus défavorable pour RECHERCHE-MÉLANGE dans les cas $k = 0$ et $k = 1$. Généralisez votre solution de façon qu'elle intègre le cas $k \geq 1$.
- i.* Quel est celui des trois algorithmes de recherche que vous utiliseriez ? Expliquez votre réponse.

NOTES

Bollobás [44], Hofri [151] et Spencer [283] contiennent une foule de techniques probabilistes avancées. Les avantages des algorithmes randomisés sont traités par Karp [174] et Rabin [253]. Le manuel de Motwani et Raghavan [228] offre un exposé très complet sur les algorithmes randomisés.

Le problème de l'embauche, à travers diverses variantes, a donné lieu à une multitude d'études. Ces problèmes sont généralement connus sous l'appellation de « problèmes des secrétaires ». Le papier de Ajtai, Meggido et Waarts [12] présente un exemple de ce genre d'étude.

PARTIE 2

TRI ET RANGS

Cette partie présente plusieurs algorithmes capables de résoudre le *problème du tri*, qu'on définit ainsi :

Entrée : Une séquence de n nombres $\langle a_1, a_2, \dots, a_n \rangle$.

Sortie : Une permutation (réarrangement) $\langle a'_1, a'_2, \dots, a'_n \rangle$ de la séquence d'entrée telle que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

La séquence d'entrée est en général un tableau à n éléments, bien qu'il puisse être également représenté différemment, par exemple sous la forme d'une liste chaînée.

Structure des données

En pratique, les nombres à trier sont rarement des valeurs isolées. Chacun fait généralement partie d'une collection de données appelée *enregistrement*. Chaque enregistrement contient une *clé*, qui est la valeur à trier, et le reste de l'enregistrement est constitué de *données satellites*, qui sont en général traitées en même temps que la clé. En pratique, lorsqu'un algorithme de tri permute les clés, il doit permuter également les données satellites. Si chaque enregistrement contient une grande quantité de données satellites, on permute souvent un tableau de pointeurs pointant vers les enregistrements, et non les enregistrements eux-mêmes, pour minimiser les déplacements de données.

En un sens, ce sont ces détails d'implémentation qui distinguent un algorithme d'un programme complet. Le fait qu'on trie des nombres isolés ou de grands enregistrements contenant des nombres n'a pas de conséquence sur la *méthode* avec laquelle

une procédure de tri détermine l'ordre de tri. Donc, quand on s'intéresse à un problème de tri, on suppose en général que l'entrée n'est constituée que de nombres. La transposition d'un algorithme triant des nombres en programme triant des enregistrements est conceptuellement immédiate, bien que dans certaines situations concrètes, on puisse rencontrer d'autres subtilités qui rendent périlleuse la tâche de programmation réelle.

Pourquoi trier ?

Nombre de théoriciens de l'informatique considèrent le tri comme le problème le plus fondamental en matière d'algorithmique. Et ce pour plusieurs raisons :

- Il peut advenir que le besoin de trier les données soit inhérent à l'application. Ainsi, pour établir les relevés bancaires de leurs clients, les banques trient les chèques par leurs numéros.
- Les algorithmes utilisent souvent le tri en tant que sous-routine vitale. Par exemple, un programme qui dessine des objets graphiques empilés les uns par dessus les autres doit trier les objets selon une relation de « recouvrement » qui lui permette de dessiner les objets en partant du bas et en remontant vers le haut. Nombre d'algorithmes de cet ouvrage emploient une sous-routine de tri.
- Il existe une foule d'algorithmes de tri, qui emploient une riche panoplie de techniques. En fait, maintes techniques majeures de l'algorithmique sont représentées sous la forme d'algorithmes de tri qui ont été développés au fil des ans. Vu de cette façon, le tri présente donc aussi un intérêt historique.
- Le tri est un problème pour lequel on peut trouver un minorant non trivial (comme nous le ferons au chapitre 8). Comme nos meilleurs majorants convergent asymptotiquement vers le minorant, nous savons que nos algorithmes de tri sont asymptotiquement optimaux. En outre, nous pouvons employer le minorant du tri pour trouver des minorants pour certains autres problèmes.
- Nombre de problèmes techniques surgissent quand on met en œuvre des algorithmes de tri. Les performances d'un programme de tri pour une situation donnée peuvent dépendre d'un grand nombre de facteurs, par exemple la connaissance que l'on a déjà concernant les clés et les données satellite, la hiérarchie des mémoires (caches et mémoire virtuelle) de l'ordinateur hôte ou l'environnement logiciel. Mieux vaut traiter ces problèmes au niveau algorithmique, plutôt que de « bidouiller » le code.

Algorithmes de tri

Au chapitre 2, nous avons présenté deux algorithmes capables de trier n nombres réels. Le tri par insertion s'exécute en $\Theta(n^2)$ dans le cas le plus défavorable. Toutefois, comme ses boucles internes sont compactes, pour les entrées de taille réduite, c'est

un algorithme rapide de tri sur place. (Comme on l'a déjà vu, un algorithme trie *sur place* quand il n'y a jamais plus d'un nombre constant d'éléments du tableau d'entrée à être stocké hors du tableau.) Le tri par fusion a un temps d'exécution asymptotique meilleur, $\Theta(n \lg n)$, mais la procédure FUSION qu'il utilise n'opère pas sur place.

Dans cette partie, nous présenterons deux algorithmes supplémentaires, qui trient des nombres réels arbitraires. Le tri par tas, étudié au chapitre 6, trie n nombres sur place en $O(n \lg n)$. Il utilise une structure de données importante, appelée tas, qui permet également d'implémenter une file de priorités.

Le tri rapide (ou quicksort) du chapitre 7, trie également n nombres sur place, mais son temps d'exécution dans le pire des cas est $\Theta(n^2)$. En revanche, son temps d'exécution moyen est $\Theta(n \lg n)$, et il bat généralement le tri par tas en pratique. À l'instar du tri par insertion, le tri rapide a un code compact, de sorte que le facteur constant implicite de son temps d'exécution est petit. C'est un algorithme très répandu pour le tri de grands tableaux.

Le tri par insertion, le tri par fusion, le tri par tas et le tri rapide sont tous des tris par comparaison : ils déterminent l'ordre d'un tableau d'entrée en comparant les éléments. Le chapitre 8 commence par présenter le modèle de l'arbre de décision pour étudier les limites de performance des tris par comparaison. À l'aide de ce modèle, on peut trouver un minorant de $\Omega(n \lg n)$ pour le temps d'exécution du cas le plus défavorable d'un tri par comparaison quelconque pour n entrées, ce qui montre que le tri par tas et le tri par fusion sont des tris par comparaison asymptotiquement optimaux.

Le chapitre 8 montre ensuite qu'on peut améliorer ce minorant de $\Omega(n \lg n)$ s'il est possible de rassembler des informations sur l'ordre des valeurs d'entrée par des moyens autres que la comparaison d'éléments. L'algorithme du tri par dénombrement, par exemple, suppose que les nombres en entrée sont dans l'ensemble $\{1, 2, \dots, k\}$. En utilisant l'indexation de tableau comme outil pour déterminer l'ordre relatif, le tri par dénombrement peut trier n nombres dans un temps $\Theta(k + n)$. Ainsi, quand $k = O(n)$, le tri par dénombrement s'exécute dans un temps qui est une fonction linéaire de la taille du tableau d'entrée. Un algorithme corollaire, le tri par base (radix sort), permet d'étendre la portée du tri par dénombrement. Si l'on a n entiers à trier, chaque entier s'écrivant sur d chiffres et chaque chiffre appartenant à l'ensemble $\{1, 2, \dots, k\}$, alors le tri par base peut trier les nombres en $\Theta(d(n + k))$. Lorsque d est une constante et k vaut $O(n)$, le tri par base s'exécute en temps linéaire. Un troisième algorithme, le tri par paquet (bucket sort), exige que l'on connaisse la distribution probabiliste des nombres dans le tableau à trier. Il peut trier n nombres réels uniformément distribués dans l'intervalle semi-ouvert $[0, 1[$ avec un temps $O(n)$ dans le cas moyen.

Rangs

Le i ème rang d'un ensemble de n nombres est le i ème plus petit nombre de l'ensemble. On peut bien sûr sélectionner le i ème rang en triant l'entrée et en indexant le i ème élément de la sortie. Si l'on ne fait pas hypothèses préalables sur la distribution

en entrée, cette méthode s'exécute en $\Omega(n \lg n)$, comme le montre le minorant établi au chapitre 8.

Au chapitre 9, on montrera qu'on peut trouver le i ème plus petit élément en $O(n)$, même quand les éléments sont des nombres réels arbitraires. Nous présenterons un algorithme avec un pseudo-code compact qui s'exécute en $\Theta(n^2)$ dans le pire des cas, et en temps linéaire en moyenne. Nous donnerons également un algorithme plus compliqué qui s'exécute en $O(n)$ dans le pire des cas.

Connaissances préliminaires

Bien que la majeure partie de cette partie ne fasse pas appel à des notions mathématiques difficiles, certaines sections ont besoin d'outils plus sophistiqués. En particulier, les analyses du cas moyen du tri rapide et du tri par paquet, ainsi que l'algorithme du rang se servent des probabilités, qui sont revues à l'annexe C. L'analyse du cas le plus défavorable de l'algorithme en temps linéaire pour le rang est plus complexe, mathématiquement parlant, que les autres analyses de cas le plus défavorable vues dans cette partie.

Chapitre 6

Tri par tas

Dans ce chapitre, nous présentons un nouvel algorithme de tri. Comme le tri par fusion, mais contrairement au tri par insertion, le temps d'exécution du tri par tas est $O(n \lg n)$. Comme le tri par insertion, mais contrairement au tri par fusion, le tri par tas trie sur place : à tout moment, jamais plus d'un nombre constant d'éléments de tableau ne se trouvera stocké hors du tableau d'entrée. Le tri par tas rassemble donc le meilleur des deux algorithmes de tri que nous avons déjà étudiés.

Le tri par tas introduit aussi une autre technique de conception des algorithmes : on utilise une structure de données, appelée ici « tas » pour gérer les informations pendant l'exécution de l'algorithme. Non seulement le tas est une structure de données utile pour le tri par tas, mais il permet aussi de construire une file de priorités efficace. Cette structure de données réapparaîtra dans des algorithmes d'autres chapitres.

Le terme « tas » fut, au départ, inventé dans le contexte du tri par tas, mais il a depuis pris le sens de « mémoire récupérable » (garbage-collected storage) comme celle fournie par les langages Lisp et Java. Pour nous, le tas n'est *pas* une portion de mémoire récupérable ; chaque fois qu'on parlera de tas dans ce livre il s'agira de la structure définie dans ce chapitre.

6.1 TAS

La structure de *tas* (*binnaire*) est un tableau qui peut être vu comme un arbre binaire presque complet (voir Section B.5.3), comme le montre la figure 6.1. Chaque nœud de l'arbre correspond à un élément du tableau qui contient la valeur du nœud. L'arbre est complètement rempli à tous les niveaux, sauf éventuellement au niveau le plus bas,

qui est rempli en partant de la gauche et jusqu'à un certain point. Un tableau A représentant un tas est un objet ayant deux attributs : $longueur[A]$, nombre d'éléments du tableau, et $taille[A]$, nombre d'éléments du tas rangés dans le tableau A . Autrement dit, bien que $A[1 .. longueur[A]]$ puisse contenir des nombres valides, aucun élément après $A[taille[A]]$, où $taille[A] \leq longueur[A]$, n'est un élément du tas. La racine de l'arbre est $A[1]$, et étant donné l'indice i d'un nœud, les indices de son parent $PARENT(i)$, de son enfant de gauche $GAUCHE(i)$ et de son enfant de droite $DROITE(i)$ peuvent être facilement calculés :

$PARENT(i)$

retourner $\lfloor i/2 \rfloor$

$GAUCHE(i)$

retourner $2i$

$DROITE(i)$

retourner $2i + 1$

Sur la plupart des ordinateurs, la procédure $GAUCHE$ peut calculer $2i$ en une seule instruction, en décalant simplement d'une position vers la gauche la représentation binaire de i . De même, la procédure $DROITE$ peut calculer rapidement $2i + 1$ en décalant d'une position vers la gauche la représentation binaire de i et en ajoutant un 1 comme bit de poids faible. La procédure $PARENT$ peut calculer $\lfloor i/2 \rfloor$ en décalant i d'une position binaire vers la droite. Dans une bonne implémentation du tri par tas, ces trois procédures sont souvent implémentées en tant que « macros », ou procédures « en ligne ».

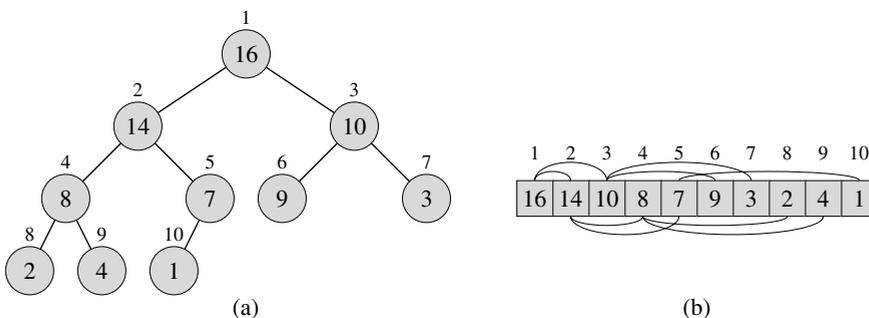


Figure 6.1 Un tas-max vu comme (a) un arbre binaire et (b) un tableau. Le nombre à l'intérieur du cercle, à chaque nœud de l'arbre, est la valeur contenue dans ce nœud. Le nombre au-dessus d'un nœud est l'indice correspondant dans le tableau. Au-dessus et au-dessous du tableau sont des lignes matérialisant les relations parent-enfant ; les parents sont toujours à gauche de leurs enfants. L'arbre a une hauteur de trois ; le nœud d'indice 4 (avec la valeur 8) a une hauteur de un.

Il existe deux sortes de tas binaires : les tas max et les tas min. Pour ces deux types de tas, les valeurs des nœuds satisfont à une *propriété de tas*, dont la nature spécifique dépend du type de tas. Dans un *tas max*, la *propriété de tas max* est que, pour chaque nœud i autre que la racine,

$$A[\text{PARENT}(i)] \geq A[i],$$

En d'autres termes, la valeur d'un nœud est au plus égale à celle du parent. Ainsi, le plus grand élément d'un tas max est stocké dans la racine, et le sous-arbre issu d'un certain nœud contient des valeurs qui ne sont pas plus grandes que celle du nœud lui-même. Un *tas min* est organisé en sens inverse ; la *propriété de tas min* est que, pour chaque nœud i autre que la racine,

$$A[\text{PARENT}(i)] \leq A[i].$$

Le plus petit élément d'un tas min est à la racine.

Pour l'algorithme du tri par tas, on utilise des tas max. Les tas min servent généralement dans les files de priorités, que nous verrons à la section 6.5. Nous précisons si nous prenons un tas max ou un tas min pour une application particulière ; pour les propriétés qui s'appliquent indifféremment aux tas max et aux tas min, nous parlerons de « tas » tout court.

En considérant un tas comme un arbre, on définit la *hauteur* d'un nœud dans un tas comme le nombre d'arcs sur le chemin simple le plus long reliant le nœud à une feuille, et on définit la hauteur du tas comme étant la hauteur de sa racine. Comme un tas de n éléments est basé sur un arbre binaire complet, sa hauteur est $\Theta(\lg n)$ (voir Exercice 6.1.2). On verra que les opérations élémentaires sur les tas s'exécutent dans un temps au plus proportionnel à la hauteur de l'arbre et prennent donc un temps $O(\lg n)$. Le reste de ce chapitre présentera quelques procédures élémentaires et montrera comment elles peuvent être utilisées dans un algorithme de tri et dans une structure de données représentant une file de priorité.

- La procédure ENTASSER-MAX, qui s'exécute en $O(\lg n)$, est la clé de voûte de la conservation de la propriété de tas max.
- La procédure CONSTRUIRE-TAS-MAX, qui s'exécute en un temps linéaire, produit un tas max à partir d'un tableau d'entrée non-ordonné.
- La procédure TRI-PAR-TAS, qui s'exécute en $O(n \lg n)$, trie un tableau sur place.
- Les procédures INSÉRER-TAS-MAX, EXTRAIRE-MAX-TAS, AUGMENTER-CLÉ-TAS et MAXIMUM-TAS, qui s'exécutent en $O(\lg n)$, permettent d'utiliser la structure de données de tas pour gérer une file de priorité.

Exercices

6.1.1 Quels sont les nombres minimal et maximal d'éléments dans un tas de hauteur h ?

6.1.2 Montrer qu'un tas à n éléments a une hauteur $\lfloor \lg n \rfloor$.

6.1.3 Montrer que pour un sous-arbre quelconque d'un tas max, la racine du sous-arbre contient la plus grande valeur parmi celles de ce sous-arbre.

6.1.4 Ou pourrait se trouver le plus petit élément d'un tas max, en supposant que tous les éléments sont distincts ?

6.1.5 Un tableau trié forme-t-il un tas min ?

6.1.6 La séquence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ forme-t-elle un tas max ?

6.1.7 Montrer que, si l'on emploie la représentation tableau pour un tas à n éléments, les feuilles sont les nœuds indexés par $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$.

6.2 CONSERVATION DE LA STRUCTURE DE TAS

ENTASSER-MAX est un sous-programme important pour la manipulation des tas max, qui prend en entrée un tableau A et un indice i . Quand ENTASSER-MAX est appelée, on suppose que les arbres binaires enracinés en $\text{GAUCHE}(i)$ et $\text{DROITE}(i)$ sont des tas max, mais que $A[i]$ peut être plus petit que ses enfants, violant ainsi la propriété de tas max. Le rôle de ENTASSER-MAX est de faire « descendre » la valeur de $A[i]$ dans le tas max de manière que le sous-arbre enraciné en i devienne un tas max.

```

ENTASSER-MAX( $A, i$ )
1   $l \leftarrow \text{GAUCHE}(i)$ 
2   $r \leftarrow \text{DROITE}(i)$ 
3  si  $l \leq \text{taille}[A]$  et  $A[l] > A[i]$ 
4      alors  $max \leftarrow l$ 
5      sinon  $max \leftarrow i$ 
6  si  $r \leq \text{taille}[A]$  et  $A[r] > A[max]$ 
7      alors  $max \leftarrow r$ 
8  si  $max \neq i$ 
9      alors échanger  $A[i] \leftrightarrow A[max]$ 
10     ENTASSER-MAX( $A, max$ )

```

La figure 6.2 illustre l'action de ENTASSER-MAX. A chaque étape, on détermine le plus grand des éléments $A[i]$, $A[\text{GAUCHE}(i)]$, et $A[\text{DROITE}(i)]$, et son indice est rangé dans max . Si $A[i]$ est le plus grand, alors le sous-arbre enraciné au nœud i est un tas max et la procédure se termine. Sinon, c'est l'un des deux enfants qui contient l'élément le plus grand, auquel cas $A[i]$ est échangé avec $A[max]$, ce qui permet au nœud i et à ses enfants de satisfaire la propriété de tas max. Toutefois, le nœud indexé par max contient maintenant la valeur initiale de $A[i]$, et donc le sous-arbre enraciné en max viole peut-être la propriété de tas max. ENTASSER-MAX doit donc être appelée récursivement sur ce sous-arbre.

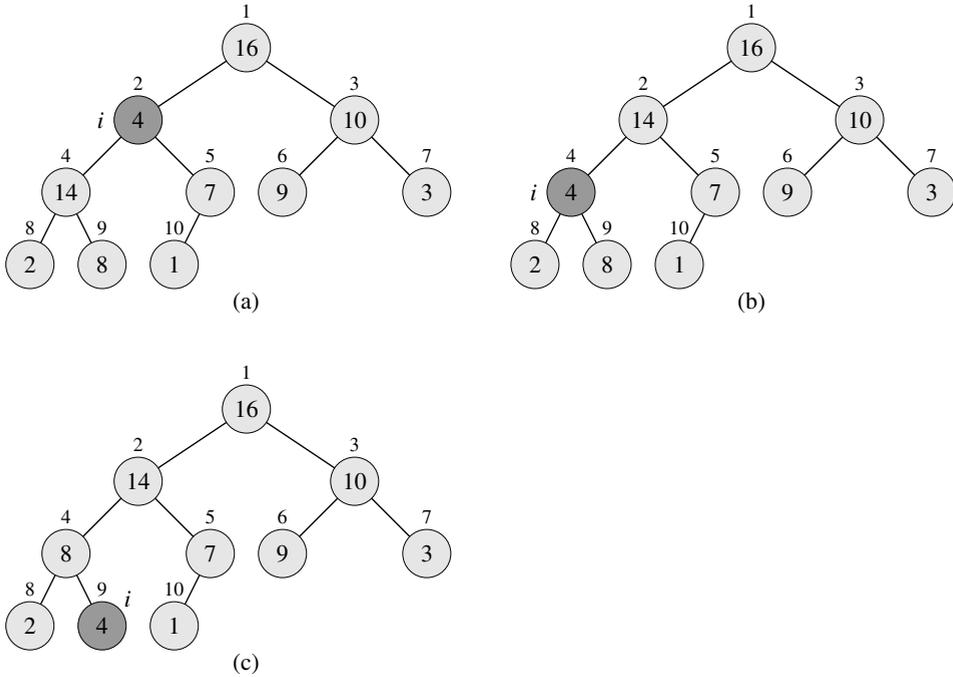


Figure 6.2 L'action de $\text{ENTASSER-MAX}(A, 2)$, où $\text{taille}[A] = 10$. (a) La configuration initiale du tas, avec la valeur $A[2]$ au nœud $i = 2$ viole la propriété de tas max puisqu'elle n'est pas supérieure à celle des deux enfants. La propriété de tas max est restaurée pour le nœud 2 en (b) via échange de $A[2]$ avec $A[4]$, ce qui détruit la propriété de tas max pour le nœud 4. L'appel récursif $\text{ENTASSER-MAX}(A, 4)$ prend maintenant $i = 4$. Après avoir échangé $A[4]$ avec $A[9]$, comme illustré en (c), le nœud 4 est corrigé, et l'appel récursif $\text{ENTASSER-MAX}(A, 9)$ n'engendre plus de modifications de la structure de données.

Le temps d'exécution de ENTASSER-MAX sur un sous-arbre de taille n enraciné en un nœud i donné est le temps $\Theta(1)$ nécessaire pour corriger les relations entre les éléments $A[i]$, $A[\text{GAUCHE}(i)]$, et $A[\text{DROITE}(i)]$, plus le temps d'exécuter ENTASSER-MAX sur un sous-arbre enraciné sur l'un des enfants du nœud i . Les sous-arbres des enfants ont chacun une taille au plus égale à $2n/3$ (le pire des cas survient quand la dernière rangée de l'arbre est remplie exactement à moitié), et le temps d'exécution de la procédure ENTASSER-MAX peut donc être décrit par la récurrence

$$T(n) \leq T(2n/3) + \Theta(1).$$

La solution de cette récurrence, d'après le cas 2 du théorème général (théorème 4.1), est $T(n) = O(\lg n)$. On peut également caractériser le temps d'exécution de ENTASSER-MAX sur un nœud de hauteur h par $O(h)$.

Exercices

6.2.1 En prenant comme modèle la figure 6.2, illustrer l'action de $\text{ENTASSER-MAX}(A, 3)$ sur le tableau $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2.2 En s’inspirant de la procédure ENTASSER-MAX, écrire du pseudo code pour la procédure ENTASSER-MIN(A, i) qui fait la même chose mais pour un tas min. Comparer le temps d’exécution de ENTASSER-MIN et celui de ENTASSER-MAX ?

6.2.3 Quel est l’effet d’un appel ENTASSER-MAX(A, i) quand l’élément $A[i]$ est plus grand que ses enfants ?

6.2.4 Quel est l’effet d’un appel ENTASSER-MAX(A, i) pour $i > \text{taille}[A]/2$?

6.2.5 Le code de ENTASSER-MAX est assez efficace en termes de facteurs constants, sauf peut-être pour l’appel récursif en ligne 10 qui risque d’entraîner certains compilateurs à générer du code inefficace. Écrire une procédure ENTASSER-MAX efficace qui utilise une structure de contrôle itérative (boucle) au lieu d’un appel récursif.

6.2.6 Montrer que le temps d’exécution dans le pire des cas de ENTASSER-MAX sur un tas de taille n est $\Omega(\lg n)$. (*conseil* : Pour un tas à n nœuds, donner des valeurs de nœuds qui provoquent l’appel récursif de ENTASSER-MAX en chaque nœud d’un chemin reliant la racine à une feuille.)

6.3 CONSTRUCTION D’UN TAS

On peut utiliser la procédure ENTASSER-MAX à l’envers pour convertir un tableau $A[1..n]$, avec $n = \text{length}[A]$, en tas max. D’après l’exercice 6.1.7, les éléments du sous-tableau $A[(\lfloor n/2 \rfloor + 1)..n]$ sont tous des feuilles de l’arbre, et donc chacun est initialement un tas à 1 élément. La procédure CONSTRUIRE-TAS-MAX parcourt les autres nœuds de l’arbre et appelle ENTASSER-MAX pour chacun.

```

CONSTRUIRE-TAS-MAX(A)
1   $\text{taille}[A] \leftarrow \text{longueur}[A]$ 
2  pour  $i \leftarrow \lfloor \text{longueur}[A]/2 \rfloor$  jusqu’à 1
3      faire ENTASSER-MAX( $A, i$ )

```

La figure 6.3 montre un exemple de l’action de CONSTRUIRE-TAS-MAX.

Pour prouver la conformité de CONSTRUIRE-TAS-MAX, on utilise l’invariant de boucle suivant : Au début de chaque itération de la boucle **pour** des lignes 2–3, chaque nœud $i + 1, i + 2, \dots, n$ est la racine d’un tas max.

Il faut montrer que cet invariant est vrai avant la première itération, que chaque itération le conserve et qu’il fournit une propriété permettant de prouver la conformité après la fin de l’exécution de la boucle.

Initialisation : Avant la première itération de la boucle, $i = \lfloor n/2 \rfloor$. Chaque nœud $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ est une feuille et est donc la racine d’un tas max trivial.

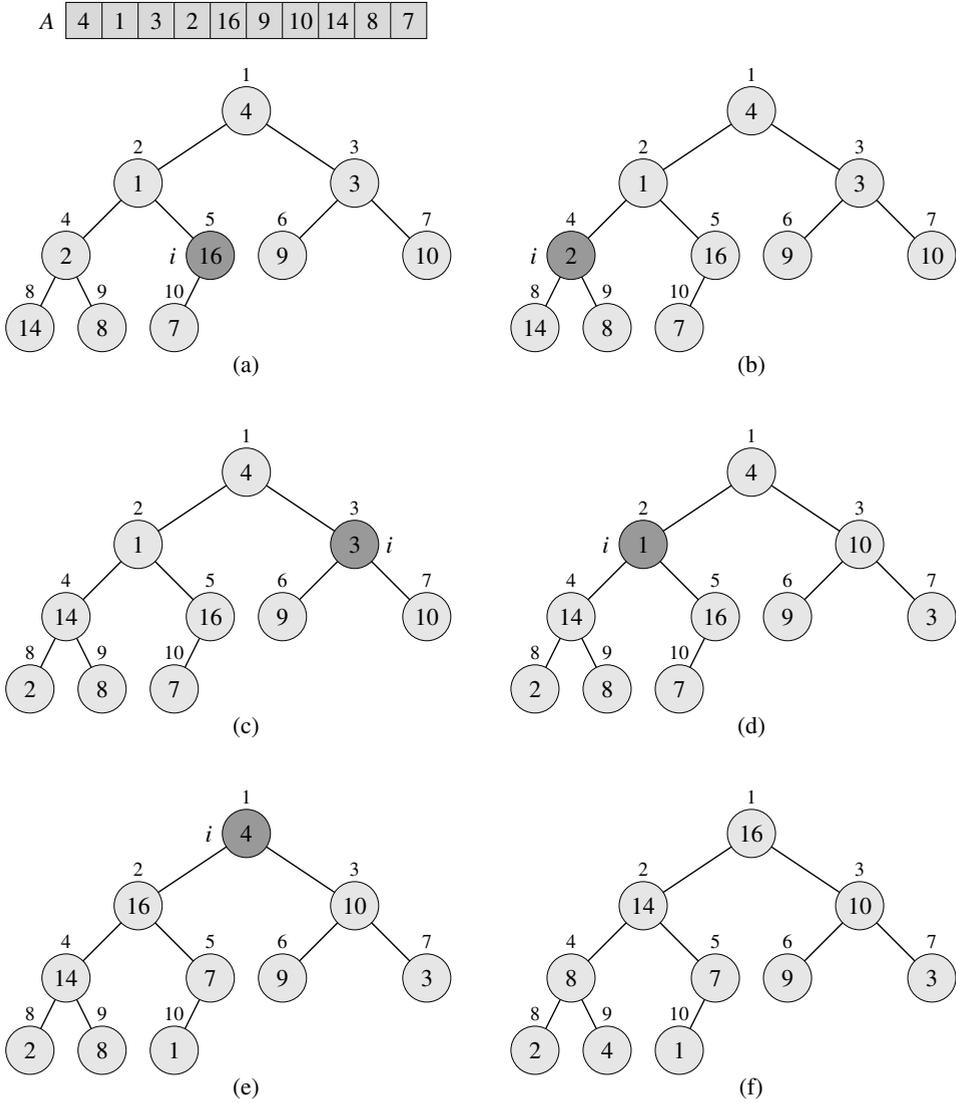


Figure 6.3 Fonctionnement de CONSTRUIRE-TAS-MAX, montrant la structure de données avant l'appel à ENTASSER-MAX en ligne 3 de CONSTRUIRE-TAS-MAX. (a) Un tableau de 10 éléments en entrée, avec l'arbre binaire qu'il représente. La figure montre que l'indice de boucle i pointe vers le nœud 5 avant l'appel ENTASSER-MAX(A, i). (b) La structure de données résultante. L'indice de boucle i de l'itération suivante pointe vers le nœud 4. (c)–(e) Les itérations suivantes de la boucle **pour** de CONSTRUIRE-TAS-MAX. Observez que, chaque fois qu'il y a appel de ENTASSER-MAX sur un nœud, les deux sous-arbres de ce nœud sont des tas max. (f) Le tas max après que CONSTRUIRE-TAS-MAX a fini.

Conservation : Pour voir que chaque itération conserve l'invariant, observez que les enfants du nœud i ont des numéros supérieurs à i . D'après l'invariant, ce sont donc tous les deux des racines de tas max. Telle est précisément la condition exigée pour que l'appel ENTASSER-MAX(A, i) fasse du nœud i une racine de tas max. En outre, l'appel ENTASSER-MAX préserve la propriété que les nœuds $i + 1, i + 2, \dots, n$ sont tous des racines de tas max. La décrémentation de i dans la partie actualisation de la boucle **pour** a pour effet de rétablir l'invariant pour l'itération suivante.

Terminaison : À la fin, $i = 0$. D'après l'invariant, chaque nœud $1, 2, \dots, n$ est la racine d'un tas max. En particulier, le nœud 1.

On peut calculer un majorant simple pour le temps d'exécution de CONSTRUIRE-TAS-MAX de la manière suivante. Chaque appel à ENTASSER-MAX coûte $O(\lg n)$, et il existe $O(n)$ appels de ce type. Le temps d'exécution est donc $O(n \lg n)$. Ce majorant, quoique correct, n'est pas asymptotiquement serré.

On peut trouver un majorant plus fin en observant que le temps d'exécution de ENTASSER-MAX sur un nœud varie avec la hauteur du nœud dans l'arbre, et que les hauteurs de la plupart des nœuds sont réduites. Notre analyse plus fine s'appuie sur les propriétés d'un tas à n éléments : sa hauteur est $\lceil \lg n \rceil$ (voir exercice 6.1.2) et le nombre de nœuds ayant la hauteur h est au plus $\lceil n/2^{h+1} \rceil$ (voir exercice 6.3.3).

Le temps requis par ENTASSER-MAX quand elle est appelée sur un nœud de hauteur h étant $O(h)$, on peut donc exprimer le coût total de CONSTRUIRE-TAS-MAX ainsi

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h} \right).$$

La dernière sommation peut être évaluée en substituant $x = 1/2$ dans la formule (A.8), ce qui donne

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2.$$

Donc, le temps d'exécution de CONSTRUIRE-TAS-MAX peut être borné par

$$\begin{aligned} O \left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h} \right) &= O \left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n). \end{aligned}$$

On peut donc construire un tas max, à partir d'un tableau non-ordonné, en un temps linéaire.

On peut construire un tas min via la procédure CONSTRUIRE-TAS-MIN qui est identique à CONSTRUIRE-TAS-MAX, sauf que l'appel à ENTASSER-MAX en ligne 3 est remplacé par un appel à ENTASSER-MIN (voir exercice 6.2.2). CONSTRUIRE-TAS-MIN produit un tas min en temps linéaire à partir d'un tableau non ordonné.

Exercices

6.3.1 En prenant modèle sur la figure 6.3, illustrer l'action de CONSTRUIRE-TAS-MAX sur le tableau $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3.2 Pourquoi fait-on décroître l'indice de boucle i de la ligne 2 de CONSTRUIRE-TAS-MAX depuis $\lfloor \text{longueur}[A]/2 \rfloor$ jusqu'à 1, au lieu de le faire croître de 1 à $\lfloor \text{longueur}[A]/2 \rfloor$?

6.3.3 * Montrer qu'il existe au plus $\lceil n/2^{h+1} \rceil$ nœuds de hauteur h dans un tas quelconque à n éléments.

6.4 ALGORITHME DU TRI PAR TAS

L'algorithme du tri par tas commence par utiliser CONSTRUIRE-TAS-MAX pour construire un tas max à partir du tableau $A[1..n]$, où $n = \text{longueur}[A]$. Comme l'élément maximal du tableau est stocké à la racine $A[1]$, on peut le placer dans sa position finale correcte en l'échangeant avec $A[n]$. Si l'on « ôte » à présent le nœud n du tas (en décrémentant $\text{taille}[A]$), on observe que $A[1..(n-1)]$ peut facilement être transformé en tas max. Les enfants de la racine restent des tas max, mais la nouvelle racine risque d'enfreindre la propriété de tas max. Pour restaurer la propriété de tas max, il suffit toutefois d'appeler une seule fois ENTASSER-MAX($A, 1$) qui laisse un tas max dans $A[1..(n-1)]$. L'algorithme du tri par tas répète alors ce processus pour le tas max de taille $n-1$ jusqu'à arriver à un tas de taille 2. (Voir exercice 6.4.2 pour plus de précisions sur l'invariant de boucle.)

```

TRI-PAR-TAS(A)
1  CONSTRUIRE-TAS-MAX(A)
2  pour  $i \leftarrow \text{longueur}[A]$  jusqu'à 2
3      faire échanger  $A[1] \leftrightarrow A[i]$ 
4           $\text{taille}[A] \leftarrow \text{taille}[A] - 1$ 
5          ENTASSER-MAX(A, 1)

```

La figure 6.4 montre l'action du tri par tas juste après la construction du tas max. Chaque tas max est montré au début d'une itération de la boucle **pour** des lignes 2–5.

La procédure TRI-PAR-TAS prend un temps $O(n \lg n)$, puisque l'appel à CONSTRUIRE-TAS-MAX prend un temps $O(n)$ et que chacun des $n-1$ appels à ENTASSER-MAX prend un temps $O(\lg n)$.

Exercices

6.4.1 En s'aidant de la figure 6.4, illustrer l'action de TRI-PAR-TAS sur le tableau $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

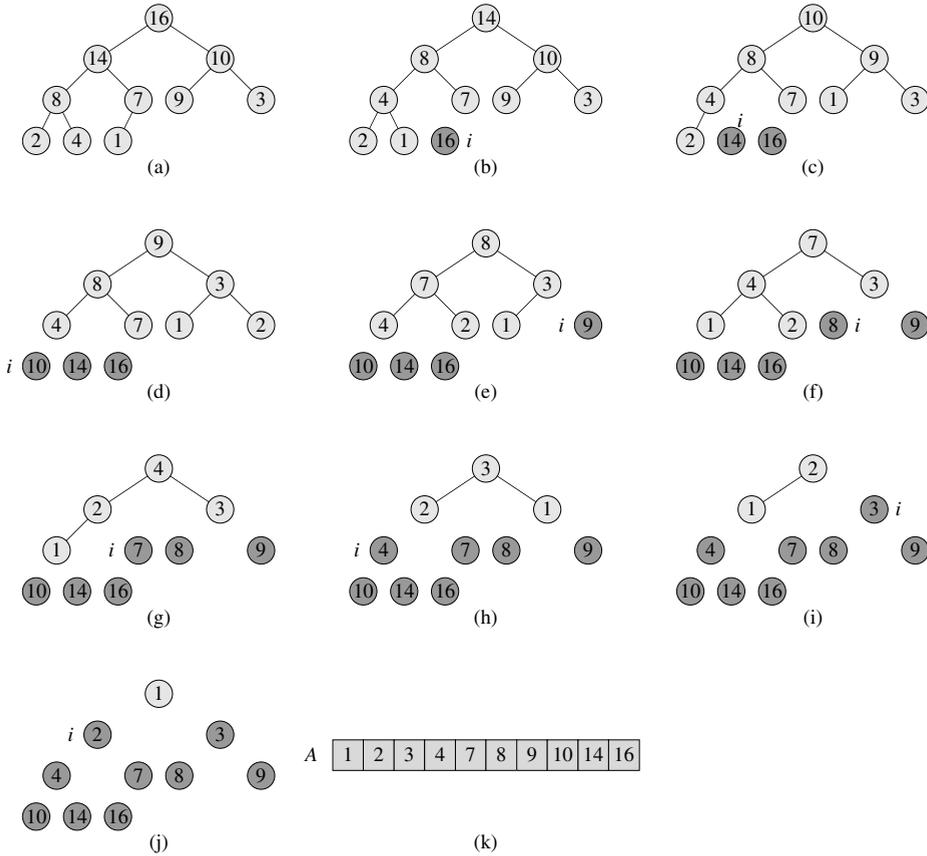


Figure 6.4 L'action de TRI-PAR-TAS. (a) La structure de tas max juste après sa construction par CONSTRUIRE-TAS-MAX. (b)–(j) Le tas max juste après chaque appel ENTASSER-MAX en ligne 5. La valeur de i à ce moment est montrée. Seul les nœuds légèrement ombrés restent dans le tas. (k) Le résultat du tri sur le tableau A .

6.4.2 Prouver la conformité de TRI-PAR-TAS à l'aide de l'invariant de boucle suivant : Au début de chaque itération de la boucle **pour** des lignes 2–5, le sous-tableau $A[1 \dots i]$ est un tas max contenant les i plus petits éléments de $A[1 \dots n]$, et le sous-tableau $A[i + 1 \dots n]$ contient les $n - i$ plus grands éléments de $A[1 \dots n]$, triés.

6.4.3 Quel est le temps d'exécution du tri par tas sur un tableau A de longueur n déjà trié en ordre croissant ? Et en ordre décroissant ?

6.4.4 Montrer que le temps d'exécution du tri par tas dans le cas le plus défavorable est $\Omega(n \lg n)$.

6.4.5 ★ Montrer que, quand tous les éléments sont distincts, le temps d'exécution optimal du tri par tas est $\Omega(n \lg n)$.

6.5 FILES DE PRIORITÉ

Le tri par tas est un excellent algorithme, mais une bonne implémentation du tri rapide (quicksort), qui sera vu au chapitre 7, est généralement plus performante en pratique. Néanmoins, la structure de données tas offre intrinsèquement de gros avantages. Dans cette section, nous allons présenter l'une des applications les plus répandues du tas, à savoir la gestion d'une file de priorité efficace. Comme c'est le cas avec les tas, il existe deux sortes de files de priorité : les files max et les files min. Nous nous concentrerons ici sur la mise en œuvre des files de priorité max, lesquelles s'appuient sur des tas max ; l'exercice 6.5.3 vous demandera d'écrire les procédures concernant les files de priorité min.

Une **file de priorité** est une structure de données qui permet de gérer un ensemble S d'éléments, dont chacun a une valeur associée baptisée **clé**. Une **file de priorité max** reconnaît les opérations suivantes.

INSÉRER(S, x) insère l'élément x dans l'ensemble S . Cette opération pourrait s'écrire sous la forme $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) retourne l'élément de S ayant la clé maximale.

EXTRAIRE-MAX(S) supprime et retourne l'élément de S qui a la clé maximale.

AUGMENTER-CLÉ(S, x, k) accroît la valeur de la clé de l'élément x pour lui donner la nouvelle valeur k , qui est censée être supérieure ou égale à la valeur courante de la clé de x .

L'une des applications des files de priorité max est de planifier les tâches sur un ordinateur. La file max gère les travaux en attente, avec leurs priorité relatives. Quand une tâche est terminée ou interrompue, l'ordinateur exécute la tâche de plus forte priorité, sélectionnée via EXTRAIRE-MAX parmi les travaux en attente. La procédure INSÉRER permet d'ajouter à tout moment une nouvelle tâche à la file.

Une **file de priorité min** reconnaît les opérations INSÉRER, MINIMUM, EXTRAIRE-MIN et DIMINUER-CLÉ. Une file de priorité min peut servir à gérer un simulateur piloté par événement. Les éléments de la file sont des événements à simuler, chacun étant affecté d'un temps d'occurrence qui lui sert de clé. Les événements doivent être simulés dans l'ordre des temps d'occurrence, vu que la simulation d'un événement risque d'entraîner la simulation ultérieure d'autres événements. Le programme de simulation emploie EXTRAIRE-MIN à chaque étape pour choisir le prochain événement à simuler. À mesure que sont produits de nouveaux événements, ils sont insérés dans la file de priorité min *via* INSÉRER. Nous verrons d'autres utilisations pour les files de priorité min, notamment l'opération DIMINUER-CLÉ, aux chapitres 23 et 24.

Il n'y a rien d'étonnant à ce qu'un tas permette de gérer une file de priorité. Dans une application donnée, par exemple dans un planificateur de tâches ou dans un simulateur piloté par événement, les éléments de la file de priorité correspondent à des objets de l'application. Il est souvent nécessaire de déterminer quel est l'objet de l'application qui correspond à un certain élément de la file de priorité, et *vice-versa*.

Quand on utilise un tas pour gérer une file de priorité, on a donc souvent besoin de stocker dans chaque élément du tas un *repère* référençant l'objet applicatif correspondant. La nature exacte du repère (pointeur ? entier ? etc.) dépend de l'application. De même, on a besoin de stocker dans chaque objet applicatif un repère référençant l'élément de tas correspondant. Ici, le repère est généralement un indice de tableau. Comme les éléments du tas changent d'emplacement dans le tableau lors des opérations de manipulation du tas, une implémentation concrète doit, en cas de relogement d'un élément du tas, actualiser en parallèle l'indice de tableau dans l'objet applicatif correspondant. Comme les détails de l'accès aux objets de l'application dépendent grandement de l'application et de son implémentation, nous n'en parlerons plus, si ce n'est pour noter que, dans la pratique, il importe de gérer correctement ces repères.

Nous allons maintenant voir comment implémenter les opérations associées à une file de priorité max. La procédure MAXIMUM-TAS implémente l'opération MAXIMUM en un temps $\Theta(1)$.

MAXIMUM-TAS(A)

1 **retourner** A[1]

La procédure EXTRAIRE-MAX-TAS implémente l'opération EXTRAIRE-MAX. Elle ressemble au corps de la boucle **pour** (lignes 3-5) de la procédure TRI-PARTAS.

EXTRAIRE-MAX-TAS(A)

1 **si** *taille*[A] < 1

2 **alors erreur** « limite inférieure dépassée »

3 *max* ← A[1]

4 A[1] ← A[*taille*[A]]

5 *taille*[A] ← *taille*[A] - 1

6 ENTASSER-MAX(A, 1)

7 **retourner** *max*

Le temps d'exécution de EXTRAIRE-MAX-TAS est $O(\lg n)$, car elle n'effectue qu'un volume constant de travail en plus du temps $O(\lg n)$ de ENTASSER-MAX.

La procédure AUGMENTER-CLÉ-TAS implémente l'opération AUGMENTER-CLÉ. L'élément de la file de priorité dont il faut accroître la clé est identifié par un indice i pointant vers le tableau. La procédure commence par modifier la clé de l'élément $A[i]$ pour lui donner sa nouvelle valeur. Accroître la clé de $A[i]$ risque d'enfreindre la propriété de tas max ; la procédure, d'une manière qui rappelle la boucle d'insertion (lignes 5-7) de TRI-INSERTION vue à la section 2.1, parcourt donc un chemin reliant ce nœud à la racine, afin de trouver une place idoine pour la clé qui vient d'être augmentée. Tout au long du parcours, elle compare un élément à son parent ; elle permute les clés puis continue si la clé de l'élément est plus grande, et elle s'arrête si la clé de l'élément est plus petite, vu que la propriété de tas max est alors satisfaite. (Voir exercice 6.5.5 pour une description précise de l'invariant de boucle.)

AUGMENTER-CLÉ-TAS($A, i, clé$)

```

1  si  $clé < A[i]$ 
2    alors erreur « nouvelle clé plus petite que clé actuelle »
3   $A[i] \leftarrow clé$ 
4  tant que  $i > 1$  et  $A[\text{PARENT}(i)] < A[i]$ 
5    faire permuter  $A[i] \leftrightarrow A[\text{PARENT}(i)]$ 
6     $i \leftarrow \text{PARENT}(i)$ 

```

La figure 6.5 montre un exemple d'opération AUGMENTER-CLÉ-TAS. Le temps d'exécution de AUGMENTER-CLÉ-TAS sur un tas à n éléments est $O(\lg n)$, car le chemin reliant le nœud, modifié en ligne 3, à la racine a la longueur $O(\lg n)$.

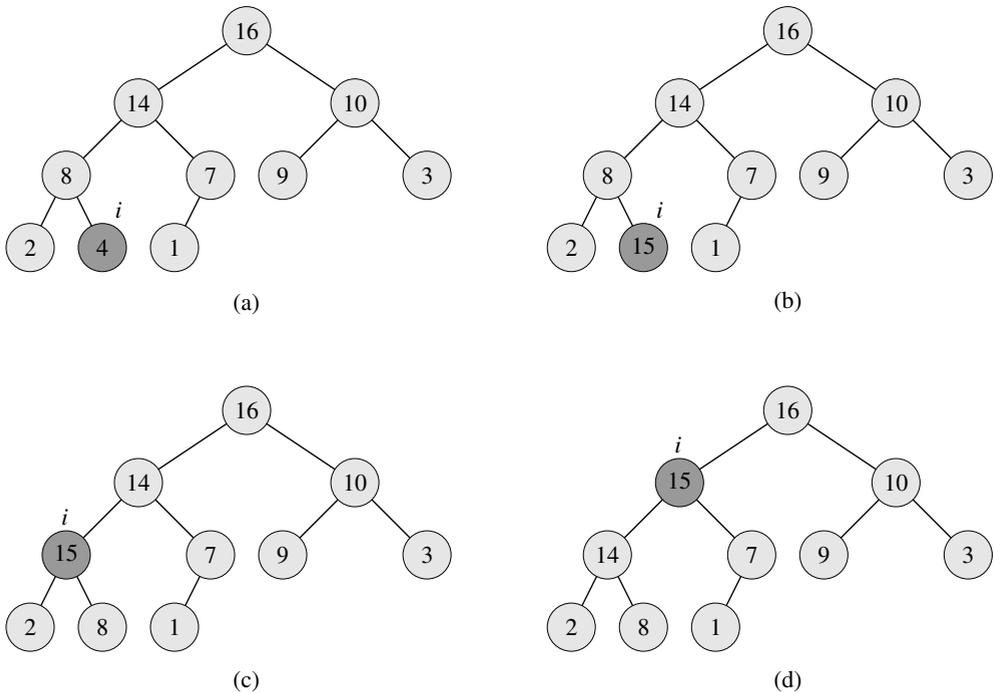


Figure 6.5 Fonctionnement de AUGMENTER-CLÉ-TAS. (a) Le tas max de la figure 6.4(a) avec un nœud dont l'indice est i en gris foncé. (b) Ce nœud voit sa clé passer à 15. (c) Après une itération de la boucle **tant que** des lignes 4–6, le nœud et son parent s'échangent leurs clés et l'indice i remonte pour devenir celui du parent. (d) Le tas max après une itération supplémentaire de la boucle **tant que**. À ce stade, $A[\text{PARENT}(i)] \geq A[i]$. La propriété de tas max étant maintenant vérifiée, la procédure se termine.

La procédure INSÉRER-TAS-MAX implémente l'opération INSÉRER. Elle prend en entrée la clé du nouvel élément à insérer dans le tas max A . La procédure commence par étendre le tas max en ajoutant à l'arbre une nouvelle feuille dont la clé est $-\infty$. Elle appelle ensuite AUGMENTER-CLÉ-TAS pour affecter à la clé du nouveau nœud la bonne valeur et conserver la propriété de tas max.

```

INSÉRER-TAS-MAX( $A, clé$ )
1   $taille[A] \leftarrow taille[A] + 1$ 
2   $A[taille[A]] \leftarrow -\infty$ 
3  AUGMENTER-CLÉ-TAS( $A, taille[A], clé$ )

```

Le temps d'exécution de INSÉRER-TAS-MAX sur un tas à n éléments est $O(\lg n)$.

En résumé, un tas permet de faire toutes les opérations de file de priorité sur un ensemble de taille n en un temps $O(\lg n)$.

Exercices

6.5.1 Illustrer le fonctionnement de EXTRAIRE-MAX-TAS sur le tas $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5.2 Illustrer le fonctionnement de INSÉRER-TAS-MAX($A, 10$) sur le tas $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$. Utiliser le tas de la figure 6.5 comme modèle pour l'appel AUGMENTER-CLÉ-TAS.

6.5.3 Écrire le pseudo code des procédures MINIMUM-TAS, EXTRAIRE-MIN-TAS, DIMINUER-CLÉ-TAS et INSÉRER-TAS-MIN qui implémentent une file de priorité min basée sur un tas min.

6.5.4 Pourquoi faut-il régler la clé du nœud inséré sur la valeur $-\infty$, en ligne 2 de INSÉRER-TAS-MAX, puisque l'étape immédiatement suivante sera de donner à la clé la valeur souhaitée ?

6.5.5 Prouver la conformité de AUGMENTER-CLÉ-TAS à l'aide de l'invariant de boucle suivant :

Au début de chaque itération de la boucle **tant que** des lignes 4–6, le tableau $A[1 \dots taille[A]]$ satisfait à la propriété de tas max, à une infraction potentielle près : $A[i]$ risque d'être plus grand que $A[PARENT(i)]$.

6.5.6 Montrer comment implémenter une file FIFO (first-in first-out) avec une file de priorité. Montrer comment implémenter une pile avec une file de priorité. (Files et piles sont définies à la section 10.1.)

6.5.7 L'opération SUPPRIMER-TAS(A, i) supprime dans le tas A l'élément placé dans le nœud i . Donner une implémentation de SUPPRIMER-TAS qui tourne en un temps $O(\lg n)$ pour un tas max à n éléments.

6.5.8 Donner un algorithme à temps $O(n \lg k)$ qui fusionne k listes triées pour produire une liste triée unique, n étant ici le nombre total d'éléments toutes listes confondues. (*Conseil* : Utiliser un tas min pour la fusion multiple.)

PROBLÈMES

6.1. Construire un tas via insertion

La procédure CONSTRUIRE-TAS-MAX de la section 6.3 peut être implémentée via un usage répété de INSÉRER-TAS-MAX pour insérer les éléments dans le tas. On considère l'implémentation suivante :

```

CONSTRUIRE-TAS-MAX'(A)
1  taille[A] ← 1
2  pour i ← 2 à longueur[A]
3      faire INSÉRER-TAS-MAX(A, A[i])
  
```

- a. Les procédures CONSTRUIRE-TAS-MAX et CONSTRUIRE-TAS-MAX' construisent-elles toujours le même tas quand elles sont exécutées sur le même tableau d'entrée ? Le démontrer, ou donner un contre-exemple.
- b. Montrer que, dans le pire des cas, CONSTRUIRE-TAS-MAX' nécessite un temps $\Theta(n \lg n)$ pour construire un tas de n éléments.

6.2. Analyse de tas d -aire

Un *tas d -aire* ressemble à un tas binaire, à ceci près qu'un nœud qui n'est pas une feuille a d enfants au lieu de 2 (à une exception potentielle près).

- a. Comment pourrait-on représenter un tas d -aire dans un tableau ?
- b. Quelle est la hauteur d'un tas d -aire à n éléments en fonction de n et d ?
- c. Donner une implémentation efficace de EXTRAIRE-MAX pour un tas max d -aire. Analyser son temps d'exécution en fonction de d et n .
- d. Donner une implémentation efficace de INSÉRER pour un tas max d -aire. Analyser son temps d'exécution en fonction de d et n .
- e. Donner une implémentation efficace de AUGMENTER-CLÉ(A, i, k), qui fait $A[i] \leftarrow \max(A[i], k)$ puis qui met à jour correctement la structure de tas max d -aire. Analyser son temps d'exécution en fonction de d et n .

6.3. Tableaux de Young

Un *tableau de Young* $m \times n$ est une matrice $m \times n$ telle que les éléments de chaque ligne sont triés dans le sens gauche-droite et les éléments de chaque colonne sont triés dans le sens haut-bas. Un tableau de Young peut contenir des valeurs ∞ , qui sont considérées comme des éléments non existants. Un tableau de Young permet donc de stocker $r \leq mn$ nombres finis.

- a. Dessiner un tableau de Young 4×4 qui contienne les éléments $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

- b. Prouver qu'un tableau de Young $m \times n$ Y est vide si $Y[1, 1] = \infty$. Prouver que Y est plein (contient mn éléments) si $Y[m, n] < \infty$.
- c. Donner un algorithme qui implémente EXTRAIRE-MIN sur un tableau de Young $m \times n$ non vide et qui tourne en un temps $O(m + n)$. L'algorithme doit utiliser une sous-routine récursive qui résout un problème $m \times n$ en résolvant récursivement soit un sous-problème $(m - 1) \times n$, soit un sous-problème $m \times (n - 1)$. (*conseil* : Penser à ENTASSER-MAX.) Soit $T(p)$, où $p = m + n$, le temps d'exécution maximal de EXTRAIRE-MIN sur un tableau de Young $m \times n$ quelconque. Donner et résoudre une récurrence pour $T(p)$ qui produise le majorant temporel $O(m + n)$.
- d. Montrer comment insérer un nouvel élément dans un tableau de Young $m \times n$ non plein en un temps $O(m + n)$.
- e. En n'utilisant pas d'autre méthode de tri comme sous-routine, montrer comment utiliser un tableau de Young $n \times n$ pour trier n^2 nombres en un temps $O(n^3)$.
- f. Donner un algorithme à temps $O(m + n)$ qui détermine si un nombre donné appartient à un tableau de Young $m \times n$ donné.

NOTES

L'algorithme du tri par tas fut inventé par Williams [316], qui a expliqué aussi comment implémenter une file de priorité à l'aide d'un tas. La procédure CONSTRUIRE-TAS-MAX a été suggérée par Floyd [90].

Nous reverrons les tas min pour implémenter des files de priorité min aux chapitres 16, 23 et 24. Nous donnerons aussi une implémentation avec des bornes temporelles améliorées pour certaines opérations aux chapitres 19 et 20.

Il est possible de trouver des implémentations plus rapides des files de priorité pour les données de type entier. Une structure de données inventée par van Emde Boas [301] permet de faire les opérations MINIMUM, MAXIMUM, INSÉRER, SUPPRIMER, RECHERCHER, EXTRAIRE-MIN, EXTRAIRE-MAX, PRÉDÉCESSEUR, et SUCCESSEUR avec un temps $O(\lg \lg C)$ dans le cas le plus défavorable, avec la restriction que l'ensemble de clés doit être l'ensemble $\{1, 2, \dots, C\}$. Si les données sont des entiers à b bits et que la mémoire de l'ordinateur se compose de mots adressables de b bits, Fredman et Willard [99] ont montré comment implémenter MINIMUM en un temps $O(1)$ et INSÉRER et EXTRAIRE-MIN en un temps $O(\sqrt{\lg n})$. Thorup [299] a amélioré la borne $O(\sqrt{\lg n})$, en la ramenant à $O((\lg \lg n)^2)$. Cette borne utilise une quantité d'espace non bornée en n , mais on peut l'implémenter en espace linéaire à l'aide d'un hachage randomisé.

Un cas particulier important des files de priorité est celui où la séquence des opérations EXTRAIRE-MIN est *monotone*, c'est-à-dire quand les valeurs produites par les appels successifs à EXTRAIRE-MIN augmentent avec le temps. Ce cas se produit dans plusieurs applications importantes, dont l'algorithme de Dijkstra pour les plus courts chemins à origine unique (que nous verrons au chapitre 24) et la simulation d'événements discrets. Pour l'algorithme de Dijkstra, il importe particulièrement que l'opération DIMINUER-CLÉ soit mise en œuvre de manière efficace. Pour le cas monotone, si les données sont des entiers appartenant à

l'intervalle $1, 2, \dots, C$, alors Ahuja, Melhorn, Orlin et Tarjan [8] expliquent comment implémenter EXTRAIRE-MIN et INSÉRER en un temps amorti $O(\lg C)$ (voir chapitre 17 pour plus de détails sur l'analyse amortie) et DIMINUER-CLÉ en un temps $O(1)$, en utilisant une structure de données dite tas base (radix heap). La borne $O(\lg C)$ peut être ramenée à $O(\sqrt{\lg C})$ en utilisant une combinaison de tas de Fibonacci (voir chapitre 20) et de tas base. La borne a été encore améliorée, passant à un temps attendu $O(\lg^{1/3+\varepsilon} C)$, par Cherkassky, Goldberg et Silverstein [58], qui combinent la structure de paquet multiniveau (multilevel bucketing) de Denardo et Fox [72] et le tas de Thorup susmentionné. Raman [256] a encore amélioré ces résultats, obtenant une borne de $O(\min(\lg^{1/4+\varepsilon} C, \lg^{1/3+\varepsilon} n))$ pour tout $\varepsilon > 0$ fixé. On trouvera des présentations détaillées de ces résultats dans des articles de Raman [256] et Thorup [299].

Chapitre 7

Tri rapide

Le tri rapide (*quicksort*) est un algorithme de tri dont le temps d'exécution, dans le cas le plus défavorable, est $\Theta(n^2)$ sur un tableau de n nombres. En dépit de ce temps d'exécution lent dans le cas le plus défavorable, le tri rapide est souvent le meilleur choix en pratique, à cause de son efficacité remarquable en moyenne : son temps d'exécution attendu est $\Theta(n \lg n)$ et les facteurs constants cachés dans la notation $\Theta(n \lg n)$ sont réduits. Il a également l'avantage de trier sur place (voir page 15) et il fonctionne bien même dans des environnements de mémoire virtuelle.

La section 7.1 décrit l'algorithme, ainsi qu'un sous-programme de partitionnement important utilisé par le tri rapide. À cause de la complexité du comportement du tri rapide, nous commencerons par une étude intuitive de ses performances dans la section 7.2 et renverrons l'analyse précise à la fin du chapitre. La section 7.3 présente une version du tri rapide qui s'appuie sur de l'échantillonnage aléatoire. Cet algorithme a un bon temps d'exécution pour le cas moyen et il n'y a aucune entrée spécifique qui entraîne le comportement le plus défavorable. L'algorithme randomisé est analysé à la section 7.4, où l'on montre qu'il s'exécute en un temps $\Theta(n^2)$ dans le cas le plus défavorable et en un temps $O(n \lg n)$ en moyenne.

7.1 DESCRIPTION DU TRI RAPIDE

Le tri rapide, comme le tri par fusion est fondé sur le paradigme diviser-pour-régner, présenté à la section 2.3.1. Voici les trois étapes du processus diviser-pour-régner employé pour trier un sous-tableau typique $A[p \dots r]$.

Diviser : Le tableau $A[p..r]$ est partitionné (réarrangé) en deux sous-tableaux (éventuellement vides) $A[p..q-1]$ et $A[q+1..r]$ tels que chaque élément de $A[p..q-1]$ soit inférieur ou égal à $A[q]$ qui, lui-même, est inférieur ou égal à chaque élément de $A[q+1..r]$. L'indice q est calculé dans le cadre de cette procédure de partitionnement.

Régner : Les deux sous-tableaux $A[p..q-1]$ et $A[q+1..r]$ sont triés par des appels récursifs au tri rapide.

Combiner : Comme les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombinaison : le tableau $A[p..r]$ tout entier est maintenant trié.

La procédure suivante implémente le tri rapide.

```

TRI-RAPIDE( $A, p, r$ )
1  si  $p < r$ 
2    alors  $q \leftarrow$  PARTITION( $A, p, r$ )
3         TRI-RAPIDE( $A, p, q - 1$ )
4         TRI-RAPIDE( $A, q + 1, r$ )

```

Pour trier un tableau A entier, l'appel initial est TRI-RAPIDE($A, 1, longueur[A]$).

a) Partitionner le tableau

Le point principal de l'algorithme est la procédure PARTITION, qui réarrange le sous-tableau $A[p..r]$ sur place.

```

PARTITION( $A, p, r$ )
1   $x \leftarrow A[r]$ 
2   $i \leftarrow p - 1$ 
3  pour  $j \leftarrow p$  à  $r - 1$ 
4    faire si  $A[j] \leq x$ 
5      alors  $i \leftarrow i + 1$ 
6          permuter  $A[i] \leftrightarrow A[j]$ 
7  permuter  $A[i + 1] \leftrightarrow A[r]$ 
8  retourner  $i + 1$ 

```

La figure 7.1 montre le fonctionnement de PARTITION sur un tableau à 8 éléments. PARTITION sélectionne toujours un élément $x = A[r]$ comme élément *pivot* autour duquel se fera le partitionnement du sous-tableau $A[p..r]$. La procédure continue et le tableau est partitionné en quatre régions (éventuellement vides). Au début de chaque itération de la boucle **pour** des lignes 3–6, chaque région satisfait à certaines propriétés qui définissent un invariant de boucle : Au début de chaque itération de la boucle des lignes 3–6, pour tout indice k ,

1) Si $p \leq k \leq i$, alors $A[k] \leq x$.

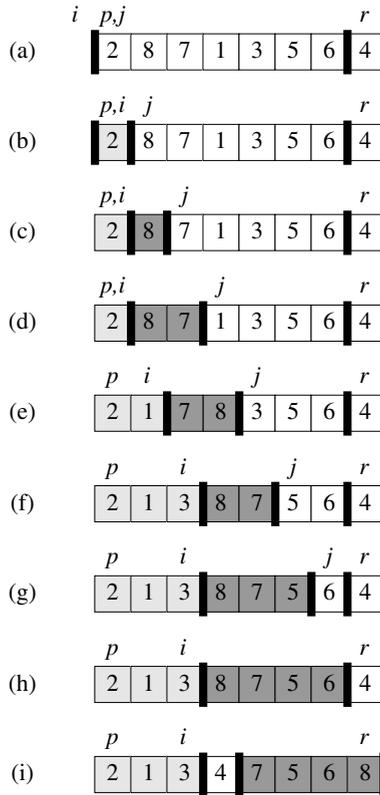


Figure 7.1 Fonctionnement de PARTITION sur un exemple. Les éléments en gris clair sont tous dans la première partition, avec des valeurs pas plus grandes que x . Les éléments en gris foncé sont dans la seconde partition, avec des valeurs supérieures à x . Les éléments non en gris n'ont pas encore été placés dans l'une des deux premières partitions et l'élément blanc final est le pivot. (a) Le tableau initial et les configurations de variable initiales. Aucun des éléments n'a été placé dans l'une quelconque des deux premières partitions. (b) La valeur 2 est « permutée avec elle-même » et placée dans la partition des petites valeurs. (c)–(d) Les valeurs 8 et 7 sont ajoutées à la partition des grandes valeurs. (e) Les valeurs 1 et 8 sont permutées et la petite partition grossit. (f) Les valeurs 3 et 7 sont échangées et la petite partition grossit. (g)–(h) La grande partition grossit pour accueillir 5 et 6 et la boucle se termine. (i) Sur les lignes 7–8, l'élément pivot est permuté de façon à aller entre les deux partitions.

2) Si $i + 1 \leq k \leq j - 1$, alors $A[k] > x$.

3) Si $k = r$, alors $A[k] = x$.

La figure 7.2 résume cette structure. Les indices entre j et $r - 1$ n'entrent dans aucun des trois cas et les valeurs des éléments correspondants n'ont pas de relation particulière avec le pivot x .

Il faut montrer que cet invariant de boucle est vrai avant la première itération, que chaque itération de la boucle conserve l'invariant et que celui-ci fournit une propriété qui prouve la conformité quand la boucle se termine.

Initialisation : Avant la première itération, $i = p - 1$ et $j = p$. Il n'y a pas de valeurs entre p et i , ni de valeurs entre $i + 1$ et $j - 1$, de sorte que les deux premières conditions de l'invariant de boucle sont satisfaites de manière triviale. L'affectation en ligne 1 satisfait à la troisième condition.

Conservation : Comme le montre la figure 7.3, il y a deux cas à considérer, selon le résultat du test en ligne 4. La figure 7.3(a) montre ce qui se passe quand $A[j] > x$; l'unique action faite dans la boucle est d'incrémenter j . Une fois j incrémenté, la condition 2 est vraie pour $A[j - 1]$ et toutes les autres éléments restent inchangés. La figure 7.3(b) montre ce qui se passe quand $A[j] \leq x$; i est incrémenté, $A[i]$ et $A[j]$ sont échangés, puis j est incrémenté. Compte tenu de la permutation, on a maintenant $A[i] \leq x$ et la condition 1 est respectée. De même, on a aussi $A[j - 1] > x$, car l'élément qui a été permuté avec $A[j - 1]$ est, d'après l'invariant de boucle, plus grand que x .

Terminaison : À la fin, $j = r$. Par conséquent, chaque élément du tableau est dans l'un des trois ensembles décrits par l'invariant et l'on a partitionné les valeurs du tableau en trois ensembles : les valeurs inférieures ou égales à x , les valeurs supérieures à x et un singleton contenant x .

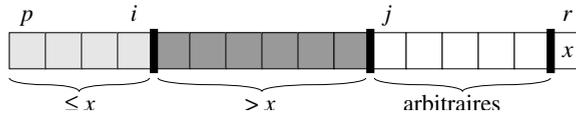


Figure 7.2 Les quatre régions gérées par la procédure PARTITION sur un sous-tableau $A[p..r]$. Les valeurs de $A[p..i]$ sont toutes inférieures ou égales à x , les valeurs de $A[i + 1..j - 1]$ sont toutes supérieures à x et $A[r] = x$. Les éléments de $A[j..r - 1]$ peuvent prendre n'importe quelles valeurs.

Les deux dernières lignes de PARTITION transfèrent le pivot vers son nouvel emplacement au milieu du tableau, en l'échangeant avec l'élément le plus à gauche qui soit supérieur à x . Le résultat de PARTITION respecte maintenant les spécifications de l'étape diviser.

Le temps d'exécution de PARTITION pour le sous-tableau $A[p..r]$ est $\Theta(n)$, où $n = r - p + 1$ (voir exercice 7.1.3).

Exercices

7.1.1 En s'inspirant de la figure 7.1, illustrer l'action de PARTITION sur le tableau $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

7.1.2 Quelle est la valeur de q retournée par PARTITION quand tous les éléments du tableau $A[p..r]$ ont la même valeur ? Modifier PARTITION pour que $q = (p + r)/2$ quand tous les éléments du tableau $A[p..r]$ ont la même valeur.

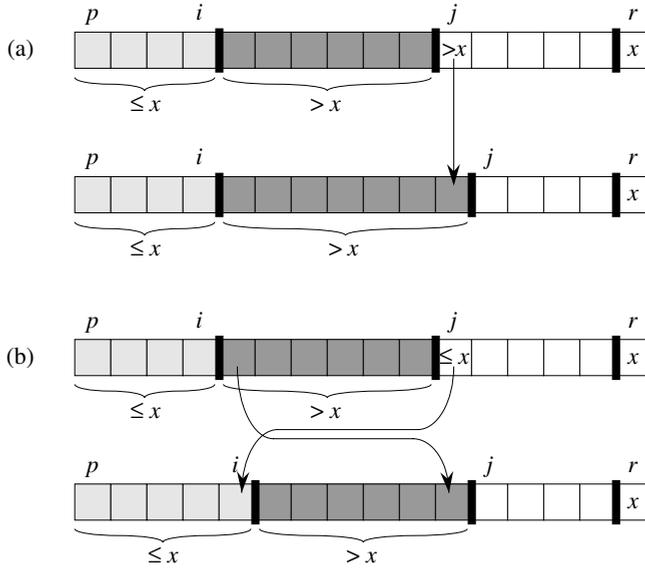


Figure 7.3 Les deux cas pour une itération de la procédure PARTITION. (a) Si $A[j] > x$, l'unique action est d'incrémenter j , ce qui conserve l'invariant de boucle. (b) Si $A[j] \leq x$, l'indice i est incrémenté, $A[i]$ et $A[j]$ sont échangés, puis j est incrémenté. Ici aussi, l'invariant de boucle est conservé.

7.1.3 Expliquer rapidement pourquoi le temps d'exécution de PARTITION sur un sous-tableau de taille n est $\Theta(n)$.

7.1.4 Comment pourrait-on modifier TRI-RAPIDE pour trier en ordre décroissant ?

7.2 PERFORMANCES DU TRI RAPIDE

Le temps d'exécution du tri rapide dépend du caractère équilibré ou non du partitionnement, qui dépend à son tour des éléments utilisés pour le partitionnement. Si le partitionnement est équilibré, l'algorithme s'exécute asymptotiquement aussi vite que le tri par fusion. En revanche, si le partitionnement n'est pas équilibré, le tri risque d'être aussi lent que le tri par insertion. Dans cette section, nous allons étudier de manière informelle les performances du tri rapide, selon que le partitionnement est équilibré ou non.

a) Partitionnement dans le cas le plus défavorable

Le cas le plus défavorable intervient pour le tri rapide quand la routine de partitionnement produit un sous-problème à $n - 1$ éléments et une autre avec 0 élément. (Cette affirmation est prouvée à la section 7.4.1.) Supposons que ce partitionnement non-équilibré survienne à chaque appel récursif. Le partitionnement coûte $\Theta(n)$. Comme

l'appel récursif sur un tableau de taille 0 rend la main sans rien faire, $T(0) = \Theta(1)$ et la récurrence pour le temps d'exécution est

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Intuitivement, si l'on cumule les coûts induits à chaque niveau de la récursivité, on obtient une série arithmétique (équation (A.2)) dont la valeur est $\Theta(n^2)$. Effectivement, la méthode de substitution prouve très simplement que la récurrence $T(n) = T(n-1) + \Theta(n)$ a pour solution $T(n) = \Theta(n^2)$. (Voir exercice 7.2.1.)

Du coup, si le partitionnement est déséquilibré au maximum à chaque niveau de récursivité de l'algorithme, le temps d'exécution est $\Theta(n^2)$. Le temps d'exécution du tri rapide n'est donc pas meilleur, dans le cas le plus défavorable, que celui du tri par insertion. En outre, ce temps d'exécution de $\Theta(n^2)$ se produit quand le tableau d'entrée est déjà complètement trié, situation courante dans laquelle le tri par insertion s'exécute en un temps $O(n)$.

b) Partitionnement dans le cas le plus favorable

Dans le partitionnement le plus équilibré possible, PARTITION produit deux sous-problèmes de taille non supérieure à $n/2$, vu que l'un est de taille $\lfloor n/2 \rfloor$ et l'autre de taille $\lceil n/2 \rceil - 1$. En pareil cas, le tri rapide s'exécute beaucoup plus rapidement. La récurrence du temps d'exécution est alors

$$T(n) \leq 2T(n/2) + \Theta(n) ,$$

D'après le cas 2 du théorème général (théorème 4.1), la solution en est $T(n) = O(n \lg n)$. Un partitionnement parfaitement équilibré à chaque niveau de la récursivité engendre donc un algorithme plus rapide asymptotiquement.

c) Partitionnement équilibré

Le temps d'exécution moyen du tri rapide est beaucoup plus près du cas optimal que du cas le plus défavorable, comme le montrera l'analyse faite à la section 7.4. Pour en comprendre la raison, il est important de comprendre comment l'équilibrage du partitionnement se répercute dans la récurrence qui décrit le temps d'exécution.

Supposons, par exemple, que l'algorithme de partitionnement produise systématiquement un découpage dans une proportion de 9 contre 1, ce qui paraît à première vue assez déséquilibré. On obtient dans ce cas la récurrence

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

pour le temps d'exécution du tri rapide ; nous avons explicitement inclus la constante c implicitement contenue dans le terme $\Theta(n)$. La figure 7.4 montre l'arbre récursif de cette récurrence. Remarquez que chaque niveau de l'arbre a un coût cn , jusqu'à ce qu'une condition aux limites soit atteinte à la profondeur $\log_{10/9} n = \Theta(\lg n)$, après

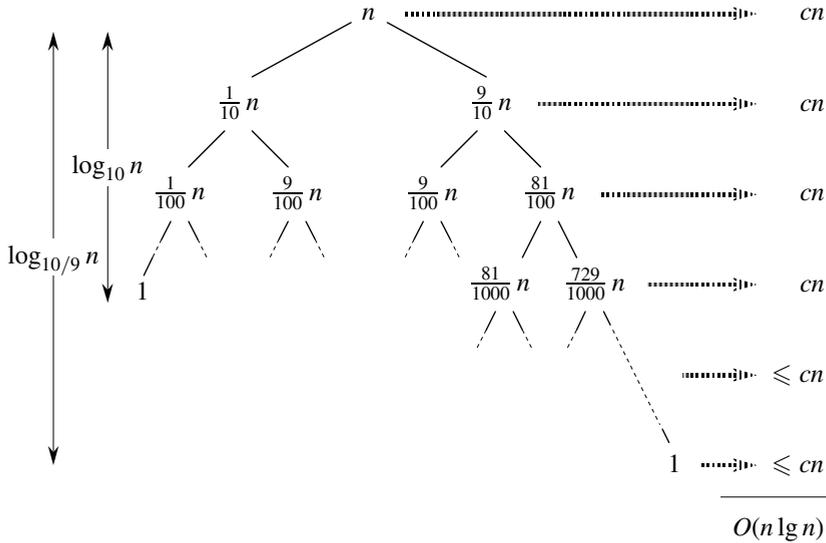


Figure 7.4 Arbre récursif de TRI-RAPIDE dans lequel PARTITION produit toujours une décomposition 9-1, donnant ainsi un temps d'exécution $O(n \lg n)$. Les nœuds montrent les tailles de sous problème, les coûts par niveau figurant à droite. Les coûts par niveau incluent la constante c implicitement contenue dans le terme $\Theta(n)$

quoi les niveaux ont un coût au plus égal à cn . La récursivité se termine à la profondeur $\log_{10/9} n = \Theta(\lg n)$. Le coût total du tri rapide est donc $O(n \lg n)$. Donc, avec un découpage 9-1 à chaque niveau de récursivité, ce qui semble intuitivement assez déséquilibré, le tri rapide s'exécute en $O(n \lg n)$, temps asymptotiquement identique à la situation où le découpage se fait exactement au milieu. En fait, même un découpage 99-1 donne un temps d'exécution $O(n \lg n)$. La raison en est qu'un découpage ayant un facteur de proportionnalité *constant* engendre toujours un arbre récursif de profondeur $\Theta(\lg n)$, où le coût de chaque niveau est $O(n)$. Le temps d'exécution est donc $O(n \lg n)$ chaque fois que le découpage s'effectue avec un facteur de proportionnalité constant.

d) Intuition pour le cas moyen

Pour se faire une idée claire du cas moyen du tri rapide, on doit faire une hypothèse sur la fréquence d'apparition escomptée des diverses entrées. Le comportement du tri rapide est conditionné par l'ordre relatif des valeurs des éléments du tableau en entrée, pas par les valeurs elles-mêmes. Comme dans notre analyse probabiliste du problème de l'embauche (voir section 5.2), nous supposons ici qu'il y a équiprobabilité pour toutes les permutations des nombres en entrée.

Lorsqu'on exécute le tri rapide sur un tableau aléatoire, il y a peu de chances pour que le partitionnement se produise toujours de la même façon à chaque niveau, comme nous l'avons supposé pour notre analyse informelle. On s'attend à ce que

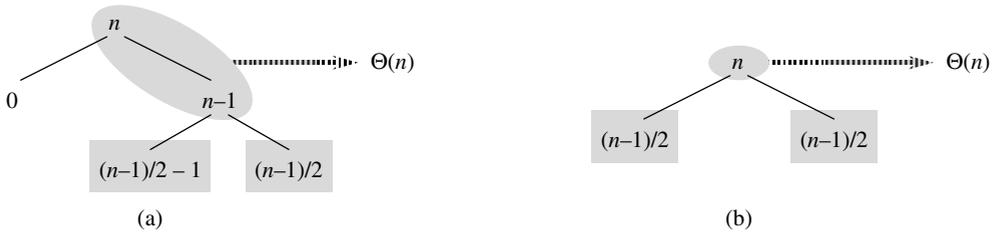


Figure 7.5 (a) Deux niveaux d'un arbre récursif du tri rapide. Le partitionnement à la racine a un coût n et produit un « mauvais » découpage : deux sous-tableaux de tailles 0 et $n - 1$. Le partitionnement du sous-tableau de taille $n - 1$ coûte $n - 1$ et produit un « bon » découpage : deux sous-tableaux de tailles $(n - 1)/2 - 1$ et $(n - 1)/2$. (b) Un niveau d'un arbre récursif qui est très bien équilibré. Dans les deux parties, le coût de partitionnement des sous-problèmes (sur fond elliptique gris) est $\Theta(n)$. Pourtant, les sous-problèmes restant à résoudre en (a), affichés sur fond carré gris, ne sont pas plus grands que les sous-problèmes correspondants qui restent à résoudre en (b).

certains découpages soient assez bien équilibrés et que d'autres soient plutôt déséquilibrés. Par exemple, l'exercice 7.2.6 vous demandera de montrer que, dans environ 80 % des cas PARTITION produit un découpage qui est plus équilibré que 9-1 et que, dans environ 20 % des cas elle produit un découpage moins équilibré que 9-1. Dans le cas moyen, PARTITION produit un mélange de « bons » et de « mauvais » découpages. Dans un arbre récursif associé du cas moyen de PARTITION, les bons et les mauvais découpages sont distribués aléatoirement tout le long de l'arbre. Supposons pourtant, pour favoriser l'intuition, que les bons et les mauvais découpages se succèdent d'un niveau de l'arbre à l'autre, que les bons découpages soient des découpages de cas optimal et que les mauvais découpages soient des découpages de cas le plus défavorable. La figure 7.5(a) montre les découpages sur deux niveaux consécutifs de l'arbre récursif. A la racine de l'arbre, le coût est n pour le partitionnement et les sous-tableaux produits ont les tailles $n - 1$ et 0 : cas le plus défavorable. Au niveau suivant, le sous-tableau de taille $n - 1$ est partitionné de façon optimale en deux sous-tableaux de tailles $(n - 1)/2 - 1$ et $(n - 1)/2$. Supposons que le coût de la condition aux limites soit 1 pour le sous-tableau de taille 0.

Le mauvais découpage suivi du bon découpage produit trois sous-tableaux de tailles 0 , $(n - 1)/2 - 1$ et $(n - 1)/2$, pour un coût total de partitionnement $\Theta(n) + \Theta(n - 1) = \Theta(n)$. Cette situation n'est certainement pas pire que celle de la figure 7.5(b), à savoir un niveau individuel de partitionnement qui produit deux sous-tableaux de tailles $(n - 1)/2$ pour un coût $\Theta(n)$. Et pourtant, cette dernière situation est équilibrée ! Intuitivement, le coût $\Theta(n - 1)$ du mauvais découpage peut être absorbé par le coût $\Theta(n)$ du bon découpage et le découpage résultant est bon. Ainsi, le temps d'exécution du tri rapide, quand les niveaux alternent entre bons et mauvais découpages, est le même que si l'on a uniquement de bons découpages : toujours $O(n \lg n)$, mais avec une constante implicite à la notation O qui est légèrement supérieure. Nous donnerons une analyse rigoureuse du cas moyen d'une version randomisée du tri rapide à la section 7.4.2.

Exercices

7.2.1 Employer la méthode de substitution pour montrer que la récurrence $T(n) = T(n-1) + \Theta(n)$ a pour solution $T(n) = \Theta(n^2)$, comme affirmé au début de la section 7.2.

7.2.2 Quel est le temps d'exécution de TRI-RAPIDE quand tous les éléments du tableau A ont la même valeur ?

7.2.3 Montrer que le temps d'exécution de TRI-RAPIDE est $\Theta(n^2)$ quand le tableau A contient des éléments distincts triés en ordre décroissant.

7.2.4 Les banques enregistrent souvent les transactions dans l'ordre chronologique, mais de nombreuses personnes aiment recevoir leurs relevés avec les chèques triés par numéros. En général, les gens signent les chèques dans l'ordre des numéros et les commerçants les encaissent dans l'ordre d'arrivée. Le problème consistant à passer d'un ordre chronologique à un ordre basé sur les numéros de chèque relève donc du problème du tri de données presque triées. Montrer que la procédure TRI-INSERTION a tendance à battre la procédure TRI-RAPIDE sur ce problème.

7.2.5 On suppose que les découpages à chaque niveau du tri rapide se font dans la proportion de $1 - \alpha$ contre α , où $0 < \alpha \leq 1/2$ est une constante. Montrer que la profondeur minimale d'une feuille de l'arbre récursif est environ $-\lg n / \lg \alpha$ et que la profondeur maximale est environ $-\lg n / \lg(1 - \alpha)$. (Ne pas se préoccuper des arrondis à la partie entière.)

7.2.6 * Montrer que, quelle que soit la constante $0 < \alpha \leq 1/2$, la probabilité est environ $1 - 2\alpha$ pour que, sur un tableau d'entrée aléatoire, PARTITION produise un découpage plus équilibré que $1 - \alpha$.

7.3 VERSIONS RANDOMISÉES DU TRI RAPIDE

Pour l'étude du comportement du tri rapide dans le cas moyen, nous avons supposé que toutes les permutations des nombres d'entrée étaient équiprobables. Dans la réalité, on ne peut pas toujours partir de cette hypothèse. (Voir exercice 7.2.4.) Comme nous l'avons vu à la section 5.3, on peut parfois ajouter de la randomisation à un algorithme pour obtenir de bonnes performances en moyenne. Beaucoup considèrent la version randomisée du tri rapide comme l'algorithme de tri à privilégier pour des entrées suffisamment grandes.

À la section 5.3, on randomisait l'algorithme en permutant explicitement l'entrée. On pourrait faire pareil pour le tri rapide, mais une autre technique de randomisation, dite *échantillonnage aléatoire*, simplifie l'analyse. Au lieu de toujours prendre $A[r]$ comme pivot, on prend un élément choisi aléatoirement dans le sous-tableau $A[p..r]$. Pour ce faire, on échange $A[r]$ avec un élément choisi au hasard dans $A[p..r]$. Cette modification, dans laquelle on échantillonne aléatoirement l'intervalle p, \dots, r , assure que l'élément pivot $x = A[r]$ a des probabilités égales d'être l'un quelconque

des $r - p + 1$ éléments du sous-tableau. Le pivot étant choisi au hasard, on peut s'attendre à ce que le partitionnement du tableau d'entrée soit, en moyenne, relativement équilibré.

Les modifications apportées à PARTITION et TRI-RAPIDE sont mineures. Dans la nouvelle procédure de partitionnement, on fait l'échange avant d'effectuer le partitionnement proprement dit :

```

PARTITION-RANDOMISE( $A, p, r$ )
1   $i \leftarrow \text{RANDOM}(p, r)$ 
2  échanger  $A[r] \leftrightarrow A[i]$ 
3  retourner PARTITION( $A, p, r$ )

```

La nouvelle version du tri rapide appelle PARTITION-RANDOMISE à la place de PARTITION :

```

TRI-RAPIDE-RANDOMISE( $A, p, r$ )
1  si  $p < r$ 
2    alors  $q \leftarrow \text{PARTITION-RANDOMISE}(A, p, r)$ 
3    TRI-RAPIDE-RANDOMISE( $A, p, q - 1$ )
4    TRI-RAPIDE-RANDOMISE( $A, q + 1, r$ )

```

Nous analyserons cet algorithme dans la prochaine section.

Exercices

7.3.1 Pourquoi analysons-nous les performances d'un algorithme randomisé pour le cas moyen et non pour le cas le plus défavorable ?

7.3.2 Pendant l'exécution de la procédure TRI-RAPIDE-RANDOMISE, combien y a-t-il d'appels au générateur de nombre aléatoires RANDOM dans le cas le plus défavorable ? Et dans le cas optimal ? Donner des réponses en terme de notation Θ .

7.4 ANALYSE DU TRI RAPIDE

La section 7.2 a donné une idée du comportement du tri rapide dans le cas le plus défavorable, ainsi que de sa rapidité d'exécution générale. Dans cette section, nous allons analyser le comportement du tri rapide de façon plus rigoureuse. Nous commencerons par une analyse du cas le plus défavorable, qui s'appliquera aussi bien à TRI-RAPIDE qu'à TRI-RAPIDE-RANDOMISE et terminerons par une analyse du cas moyen de TRI-RAPIDE-RANDOMISE.

7.4.1 Analyse du cas le plus défavorable

Nous avons vu, à la section 7.2, qu'un découpage le plus défavorable à chaque niveau de récursivité engendre un temps d'exécution $\Theta(n^2)$ qui, intuitivement, est le temps d'exécution de l'algorithme dans le cas le plus défavorable. Nous allons à présent démontrer cette assertion.

A l'aide de la méthode de substitution (voir section 4.1), on peut montrer que le temps d'exécution du tri rapide est $O(n^2)$. Soit $T(n)$ le temps d'exécution de la procédure TRI-RAPIDE dans le cas le plus défavorable sur une entrée de taille n . On a la récurrence

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n), \quad (7.1)$$

où le paramètre q est dans l'intervalle 0 à $n - 1$, puisque la procédure PARTITION génère deux sous-problèmes de taille totale $n - 1$. Nous subodorons que $T(n) \leq cn^2$ pour une certaine constante c . En intégrant cette conjecture à la récurrence (7.1), nous avons

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n). \end{aligned}$$

L'expression $q^2 + (n - q - 1)^2$ atteint un maximum en chaque extrémité de l'intervalle de paramètre $0 \leq q \leq n - 1$, car la dérivée seconde de l'expression par rapport à q est positive (voir exercice 7.4.3). Cette observation fournit la borne

$$\max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1.$$

En continuant notre majoration de $T(n)$, nous obtenons

$$\begin{aligned} T(n) &\leq cn^2 - c(2n - 1) + \Theta(n) \\ &\leq cn^2, \end{aligned}$$

vu que nous pouvons choisir la constante c suffisamment grande pour que le terme $c(2n - 1)$ domine le terme $\Theta(n)$. Donc, $T(n) = O(n^2)$. Nous avons vu, à la section 7.2, un cas spécial où le tri rapide prend un temps $\Omega(n^2)$: quand le partitionnement est déséquilibré. À titre d'alternative, l'exercice 7.4.1 vous demandera de montrer que la récurrence (7.1) a une solution de $T(n) = \Omega(n^2)$. Ainsi, le temps d'exécution (le plus défavorable) du tri rapide est $\Theta(n^2)$.

7.4.2 Temps d'exécution attendu

Nous avons déjà donné une justification intuitive du fait que le temps d'exécution de TRI-RAPIDE-RANDOMISE, dans le cas moyen, est $O(n \lg n)$: si, à chaque niveau de récursivité, le découpage induit par PARTITION-RANDOMISE place une fraction constante des éléments sur un côté de la partition, alors l'arbre récursif a une profondeur $\Theta(\lg n)$ et il y a exécution en $O(n)$ à chaque niveau. Même si l'on ajoute de

nouveaux niveaux avec le découpage le plus déséquilibré qui soit entre ces niveaux, le temps total sera toujours $O(n \lg n)$. On peut analyser, de manière très précise, l'espérance du temps d'exécution de TRI-RAPIDE-RANDOMISE en commençant par comprendre le fonctionnement de la procédure de partitionnement, puis en utilisant cette connaissance pour déduire une borne $O(n \lg n)$ pour l'espérance du temps d'exécution. Ce majorant du temps d'exécution attendu, combiné avec la borne $\Theta(n \lg n)$ du cas optimal vue à la section 7.2, donne un temps d'exécution attendu de $\Theta(n \lg n)$.

a) Temps d'exécution et comparaisons

Le temps d'exécution de TRI-RAPIDE est dominé par le temps consommé dans la procédure PARTITION. Chaque fois que PARTITION est appelée, il y a sélection d'un élément pivot ; cet élément ne figurera jamais dans les appels récursifs suivants à TRI-RAPIDE et à PARTITION. Il ne peut donc y avoir que n appels au plus à PARTITION pendant l'exécution de l'algorithme du tri rapide. Un appel à PARTITION prend $O(1)$, plus un temps qui est proportionnel au nombre d'itérations de la boucle **pour** des lignes 3–6. Chaque itération de cette boucle effectue une comparaison en ligne 4, comparant le pivot à un autre élément du tableau A . Par conséquent, si nous pouvons compter le nombre total de fois que la ligne 4 est exécutée, alors nous pourrions borner le temps total consommé dans la boucle **pour** lors de l'exécution de TRI-RAPIDE.

Lemme 7.1 *Soit X le nombre de comparaisons effectuées sur la ligne 4 de PARTITION pendant toute l'exécution de TRI-RAPIDE pour un tableau à n éléments. Alors, le temps d'exécution de TRI-RAPIDE est $O(n + X)$.*

Démonstration : Il ressort de la discussion précédente qu'il y a n appels à PARTITION, dont chacun fait un volume constant de travail puis exécute la boucle **pour** un certain nombre de fois. Chaque itération de la boucle **pour** exécute la ligne 4. \square

Il nous faut donc calculer X , nombre total de comparaisons effectuées sur l'ensemble des appels à PARTITION. Nous n'essaierons pas d'analyser le nombre de comparaisons qui sont faites dans *chaque* appel à PARTITION. Nous allons plutôt déterminer une borne globale pour le nombre total de comparaisons. Pour ce faire, nous devons comprendre dans quel cas l'algorithme compare deux éléments du tableau et dans quel cas il ne les compare pas. Pour simplifier l'analyse, renommons les éléments du tableau A comme z_1, z_2, \dots, z_n , où z_i est le i -ème plus petit élément. Nous définissons aussi $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$ comme étant l'ensemble des éléments compris entre z_i et z_j inclus.

Quand est-ce que l'algorithme compare z_i et z_j ? Pour répondre à cette question, observons d'abord que chaque paire d'éléments est testée une fois au plus. Pourquoi donc ? Les éléments ne sont comparés qu'au pivot ; or, après que l'appel à PARTITION s'est terminé, le pivot employé dans cet appel n'est plus jamais comparé aux autres éléments.

Notre analyse utilise des variables indicatrices (voir section 5.2). Soit

$$X_{ij} = I \{z_i \text{ est comparé à } z_j\} ,$$

Nous voulons savoir si la comparaison a lieu à n'importe quel moment pendant l'exécution de l'algorithme, et pas seulement pendant une itération ou un appel individuel à PARTITION. Comme chaque paire n'est testée qu'une fois au plus, on peut facilement caractériser le nombre total de comparaisons effectuées par l'algorithme :

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} .$$

En prenant les espérances des deux côtés, puis en utilisant la linéarité de l'espérance et le lemme 5.1, on obtient

$$\begin{aligned} E[X] &= E \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr \{z_i \text{ est comparé à } z_j\} . \end{aligned} \quad (7.2)$$

Reste à calculer $\Pr \{z_i \text{ est comparé à } z_j\}$.

Il est utile de se demander dans quel cas il n'y a *pas* comparaison de deux éléments. Supposons que le tri rapide doit trier les nombres 1 à 10 (rangés dans n'importe quel ordre) et que le premier pivot soit 7. Alors, le premier appel à PARTITION divise les nombres en deux ensembles : $\{1, 2, 3, 4, 5, 6\}$ et $\{8, 9, 10\}$. Ce faisant, le pivot 7 est comparé à tous les autres éléments, mais aucun nombre du premier ensemble (par exemple 2) n'est jamais comparé à un nombre quelconque du second ensemble (par exemple 9).

En général, une fois choisi un pivot x tel que $z_i < x < z_j$, on sait que z_i et z_j ne seront jamais comparés par la suite. Si, en revanche, z_i est choisi comme pivot avant tout autre élément de Z_{ij} , alors z_i sera comparé à chaque élément de Z_{ij} , sauf à lui-même. De même, si z_j est choisi comme pivot avant tout autre élément de Z_{ij} , alors z_j sera comparé à chaque élément de Z_{ij} , sauf à lui-même. Dans notre exemple, les valeurs 7 et 9 sont comparées car 7 est le premier élément de $Z_{7,9}$ à être choisi comme pivot. En revanche, 2 et 9 ne seront jamais comparés parce que le premier pivot choisi dans $Z_{2,9}$ est 7. Donc, z_i et z_j sont comparés si et seulement si le premier élément à être choisi comme pivot dans Z_{ij} est z_i ou z_j .

Calculons maintenant la probabilité de cet événement. Avant qu'un élément de Z_{ij} ne soit choisi comme pivot, tout l'ensemble Z_{ij} est dans la même partition. Donc, chaque élément de Z_{ij} a la même chance d'être le premier élément choisi comme

pivot. Comme Z_{ij} a $j - i + 1$ éléments et que les pivots sont choisis aléatoirement et de manière indépendante, la probabilité qu'un élément donné soit le premier à être choisi comme pivot est $1/(j - i + 1)$. On a donc

$$\begin{aligned}
 \Pr \{z_i \text{ est comparé à } z_j\} &= \Pr \{z_i \text{ ou } z_j \text{ est le premier pivot choisi dans } Z_{ij}\} \\
 &= \Pr \{z_i \text{ est le premier pivot choisi dans } Z_{ij}\} \\
 &\quad + \Pr \{z_j \text{ est le premier pivot choisi dans } Z_{ij}\} \\
 &= \frac{1}{j - i + 1} + \frac{1}{j - i + 1} \\
 &= \frac{2}{j - i + 1}. \tag{7.3}
 \end{aligned}$$

La deuxième ligne découle de ce que les deux événements sont mutuellement exclusifs. En combinant les équations (7.2) et (7.3), on obtient

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1}.$$

On peut calculer cette somme à l'aide d'un changement de variables ($k = j - i$) et d'une borne de la série harmonique (équation (A.7)) :

$$\begin{aligned}
 E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \\
 &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} \\
 &< \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\
 &= \sum_{i=1}^{n-1} O(\lg n) \\
 &= O(n \lg n). \tag{7.4}
 \end{aligned}$$

En conclusion, en employant PARTITION-RANDOMISE on arrive à un temps d'exécution attendu de $O(n \lg n)$ pour le tri rapide.

Exercices

7.4.1 Montrer que, dans la récurrence

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n),$$

$$T(n) = \Omega(n^2).$$

7.4.2 Montrer que le temps d'exécution du tri rapide, dans le meilleur des cas, est $\Omega(n \lg n)$.

7.4.3 Montrer que $q^2 + (n - q - 1)^2$ atteint un maximum sur l'intervalle $q = 0, 1, \dots, n - 1$ quand $q = 0$ ou $q = n - 1$.

7.4.4 Montrer que le temps d'exécution attendu de TRI-RAPIDE-RANDOMISE est $\Omega(n \lg n)$.

7.4.5 Le temps d'exécution du tri rapide peut être amélioré, en pratique, en tirant avantage du temps d'exécution efficace du tri par insertion lorsque l'entrée est « presque » triée. Lorsque le tri rapide est appelé sur un sous-tableau ayant moins de k éléments, il rend la main sans trier le sous-tableau. Après que l'appel de premier niveau du tri rapide a rendu la main, on lance le tri par insertion sur le tableau entier pour terminer le traitement. Démontrer que cet algorithme s'exécute avec un temps attendu $O(nk + n \lg(n/k))$. Comment faut-il choisir k , en théorie et en pratique ? Soit la modification suivante de la procédure PARTITION : on choisit au hasard trois éléments du tableau A , puis on effectue le partitionnement autour de l'élément médian (valeur du milieu). Approximer la probabilité d'obtenir au pire un découpage α - $(1 - \alpha)$, en tant que fonction de α dans l'intervalle $0 < \alpha < 1$.

PROBLÈMES

7.1. Validité du partitionnement de Hoare

La version de PARTITION donnée dans ce chapitre n'est pas l'algorithme de partitionnement originel. Voici la version d'origine, due à C.A.R. Hoare :

```

HOARE-PARTITION( $A, p, r$ )
1   $x \leftarrow A[p]$ 
2   $i \leftarrow p - 1$ 
3   $j \leftarrow r + 1$ 
4  tant que VRAI
5      faire répéter  $j \leftarrow j - 1$ 
6          jusqu'à  $A[j] \leq x$ 
7      répéter  $i \leftarrow i + 1$ 
8          jusqu'à  $A[i] \geq x$ 
9      si  $i < j$ 
10         alors échanger  $A[i] \leftrightarrow A[j]$ 
11         sinon retourner  $j$ 
    
```

- a. Montrer le bon fonctionnement de HOARE-PARTITION sur le tableau $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, en donnant les valeurs du tableau et les valeurs auxiliaires après chaque itération de la boucle **tant que** des lignes 4–11.

Les trois questions suivantes vont vous donner l'occasion de démontrer très précisément que la procédure HOARE-PARTITION est correcte. Prouver les points suivants :

- b. Les indices i et j sont tels que l'on n'accède jamais à un élément de A qui soit en dehors du sous-tableau $A[p..r]$.
- c. Quand HOARE-PARTITION se termine, elle retourne une valeur j telle que $p \leq j < r$.
- d. Chaque élément de $A[p..j]$ est inférieur ou égal à chaque élément de $A[j+1..r]$ quand HOARE-PARTITION se termine.

La procédure PARTITION de la section 7.1 sépare le pivot (originellement en $A[r]$) des deux partitions qu'elle crée. La procédure HOARE-PARTITION, en revanche, place toujours le pivot (originellement en $A[p]$) dans l'une des deux partitions $A[p..j]$ et $A[j+1..r]$. Comme $p \leq j < r$, ce découpage n'est jamais trivial.

- e. Réécrire la procédure TRI-RAPIDE pour qu'elle utilise HOARE-PARTITION.

7.2. Autre analyse du tri rapide

Une autre façon d'analyser le temps d'exécution du tri rapide randomisé s'appuie sur le temps d'exécution attendu de chaque appel récursif à TRI-RAPIDE, et non sur le nombre de comparaisons effectuées.

- a. Prouver que, étant donné un tableau de taille n , la probabilité qu'un quelconque élément soit choisi comme pivot est $1/n$. Utilisez ce fait pour définir des variables aléatoires indicatrices $X_i = I\{\text{le } i\text{-ème plus petit élément est choisi comme pivot}\}$. Que vaut $E[X_i]$?
- b. Soit $T(n)$ une variable aléatoire désignant le temps d'exécution du tri rapide sur un tableau de taille n . Prouver que

$$E[T(n)] = E\left[\sum_{q=1}^n X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \quad (7.5)$$

- c. Montrer que l'équation (7.5) peut être réécrite sous la forme

$$E[T(n)] = \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + \Theta(n). \quad (7.6)$$

- d. Montrer que

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2. \quad (7.7)$$

(conseil : Diviser la somme en deux parties, une pour $k = 2, 3, \dots, \lceil n/2 \rceil - 1$ et l'autre pour $k = \lceil n/2 \rceil, \dots, n - 1$.)

- e. En utilisant la borne établie dans l'équation (7.7), montrer que la récurrence dans l'équation (7.6) a pour solution $E[T(n)] = \Theta(n \lg n)$. (Conseil : Montrer, par substitution, que $E[T(n)] \leq an \log n$, pour n suffisamment grand et pour une certaine constante positive a .)

7.3. Le tri faire-valoir

Les professeurs Croquignol, Ribouldingue et Filochard proposent l’algorithme « élégant » que voici :

```

TRI-FAIRE-VALOIR( $A, i, j$ )
1  si  $A[i] > A[j]$ 
2    alors échanger  $A[i] \leftrightarrow A[j]$ 
3  si  $i + 1 \geq j$ 
4    alors retourner
5   $k \leftarrow \lfloor (j - i + 1)/3 \rfloor$            ▷ arrondi inférieur.
6  TRI-FAIRE-VALOIR( $A, i, j - k$ )         ▷ deux premiers tiers.
7  TRI-FAIRE-VALOIR( $A, i + k, j$ )       ▷ deux derniers tiers.
8  TRI-FAIRE-VALOIR( $A, i, j - k$ )       ▷ deux premiers tiers derechef.
    
```

- Démontrer que TRI-FAIRE-VALOIR($A, 1, longueur[A]$) trie correctement le tableau $A[1 \dots n]$, si $n = longueur[A]$.
- Donner une récurrence pour le temps d’exécution, dans le pire des cas, de TRI-FAIRE-VALOIR et une borne asymptotique serrée (en notation Θ) pour le temps d’exécution du cas le plus défavorable.
- Comparer le temps d’exécution de TRI-FAIRE-VALOIR, dans le cas le plus défavorable, avec les temps d’exécution du tri par insertion, du tri par fusion, du tri par tas et du tri rapide. Les professeurs méritent-ils leur titre ?

7.4. Profondeur de pile du tri rapide

L’algorithme TRI-RAPIDE de la section 7.1 contient deux appels récursifs à lui-même. Après l’appel à PARTITION, le sous-tableau de gauche est trié récursivement, puis le sous-tableau de droite est trié récursivement. Le second appel récursif à TRI-RAPIDE n’est pas vraiment nécessaire ; on peut l’éviter en utilisant une structure de contrôle itérative. Cette technique, appelée *récursivité terminale*, est utilisée automatiquement par les bons compilateurs. On considère la version suivante du tri rapide, qui simule la récursivité terminale.

```

TRI-RAPIDE'(A, p, r)
1  tant que  $p < r$ 
2    faire ▷ Partitionne et trie sous-tableau gauche.
3       $q \leftarrow \text{PARTITION}(A, p, r)$ 
4      TRI-RAPIDE'(A, p, q - 1)
5       $p \leftarrow q + 1$ 
    
```

- Démontrer que TRI-RAPIDE'(A, 1, longueur[A]) trie correctement le tableau A.

Les compilateurs exécutent généralement les procédures récursives en utilisant une *pile* qui contient les informations pertinentes, notamment les valeur des paramètres,

pour chaque appel récursif. Les données de l'appel le plus récent se trouvent au sommet de la pile, et celles du premier appel sont en bas. Lorsqu'une procédure est invoquée, ses informations sont *empilées* ; lorsqu'elle se termine, elles sont *dépilées*. Comme on suppose que les paramètres tableau sont représentés par des pointeurs, les données de chaque appel de procédure nécessitent un espace de pile $O(1)$. La *profondeur de pile* est la quantité maximale d'espace de pile utilisée lors d'un calcul.

- b. Décrire un scénario dans lequel la profondeur de pile de TRI-RAPIDE est $\Theta(n)$ sur un tableau à n éléments.
- c. Modifier le code de TRI-RAPIDE pour que la profondeur de pile, dans le cas le plus défavorable, soit $\Theta(\lg n)$. Conserver le temps d'exécution attendu de $O(n \lg n)$ pour l'algorithme.

7.5. Partition autour du nombre médian

Une façon d'améliorer la procédure TRI-RAPIDE-RANDOMISE est d'effectuer le partitionnement autour d'un pivot qui est choisi d'une manière plus fine que la sélection aléatoire d'un élément dans le sous-tableau. Une approche fréquente est la méthode du *médian de 3* : on choisit comme pivot le médian (élément du milieu) d'un ensemble de trois éléments sélectionnés aléatoirement dans le sous-tableau. (Voir exercice 7.4.6.) Pour ce problème, on suppose que les éléments du tableau d'entrée $A[1..n]$ sont distincts et que $n \geq 3$. On appelle $A'[1..n]$ le tableau trié en sortie. En utilisant la méthode du médian de 3 pour choisir le pivot, définir $p_i = \Pr\{x = A'[i]\}$.

- a. Donner une formule exacte pour p_i , sous la forme d'une fonction de n et de i pour $i = 2, 3, \dots, n-1$. (Notez que $p_1 = p_n = 0$.)
- b. De combien améliore-t-on la probabilité de choisir comme pivot $x = A'[(n+1)/2]$, médian de $A[1..n]$, par rapport à l'implémentation ordinaire ? On supposera que $n \rightarrow \infty$ et on donnera le rapport limite de ces probabilités.
- c. Si l'on définit un « bon » découpage comme signifiant le choix comme pivot de $x = A'[i]$, où $n/3 \leq i \leq 2n/3$, de combien a-t-on amélioré les chances d'obtenir un bon découpage, par rapport à l'implémentation ordinaire ? (*Conseil* : Faire une approximation de la somme par une intégrale.)
- d. Prouver que, dans le temps d'exécution $\Omega(n \lg n)$ du tri rapide, la méthode du médian de 3 n'affecte que le facteur constant.

7.6. Tri flou d'intervalles

On considère un problème de tri dans lequel les nombres ne sont pas connus avec précision : pour chacun des nombres, on connaît un intervalle de réels auquel appartient le nombre. En d'autres termes, on a n intervalles fermés de la forme $[a_i, b_i]$, avec $a_i \leq b_i$. Le but est de faire un *tri flou* de ces intervalles, c'est à dire de produire une permutation $\langle i_1, i_2, \dots, i_n \rangle$ des intervalles, telle qu'il existe $c_j \in [a_{i_j}, b_{i_j}]$ qui satisfasse à $c_1 \leq c_2 \leq \dots \leq c_n$.

- a. Concevoir un algorithme pour le tri flou de n intervalles. L'algorithme devra avoir l'allure générale d'un algorithme qui fait du tri rapide sur les extrémités gauche (les a_i), mais il devra exploiter les recouvrements d'intervalles pour améliorer le temps d'exécution. (Quand les intervalles se recourent de plus en plus, le problème du tri flou devient de plus en plus facile. L'algorithme devra utiliser les recouvrements, pour autant qu'il y en ait.)
- b. Prouver que votre algorithme tourne avec le temps attendu $\Theta(n \lg n)$ en général, mais avec le temps attendu $\Theta(n)$ quand tous les intervalles se recourent (c'est-à-dire, quand il existe une valeur x telle que $x \in [a_i, b_i]$ pour tout i). Votre algorithme ne doit pas tester ce cas explicitement ; il faut plutôt que les performances augmentent naturellement à mesure qu'augmente le volume des recouvrements.

NOTES

Le tri rapide a été inventé par Hoare [147] ; la version de Hoare apparaît au problème 7.1. La procédure PARTITION de la section 7.1 est due à N. Lomuto. L'analyse faite à la section 7.4 est due à Avrim Blum. Sedgewick [268] et Bentley [40] sont de bonnes références sur les détails d'implémentation et leur importance.

McIlroy [216] a montré comment créer un « adversaire tueur » qui produit un tableau pour lequel presque toutes les implémentations du tri rapide prennent un temps $\Theta(n^2)$. Si l'implémentation est randomisée, l'adversaire produit le tableau après avoir vu les choix aléatoires de l'algorithme de tri rapide.

Chapitre 8

Tri en temps linéaire

Nous avons présenté jusqu'ici plusieurs algorithmes capables de trier n nombres en $O(n \lg n)$. Le tri par fusion et le tri par tas atteignent ce majorant dans le cas le plus défavorable ; pour le tri rapide, cette borne est atteinte dans le cas moyen. Par ailleurs, on peut fournir à chacun de ces algorithmes une séquence de n nombres qui provoque l'exécution de l'algorithme dans un temps $\Omega(n \lg n)$.

Ces algorithmes ont en commun une propriété intéressante : *le tri qu'ils effectuent repose uniquement sur des comparaisons entre les éléments d'entrée*. Ces algorithmes de tri sont appelés **tris par comparaison**. Tous les algorithmes de tri étudiés jusqu'ici sont des tris par comparaison.

Dans la section 8.1, nous démontrerons qu'un tri par comparaison doit effectuer au pire $\Omega(n \lg n)$ comparaisons pour trier n éléments. Le tri par fusion et le tri par tas sont donc asymptotiquement optimaux et il n'existe aucun tri par comparaison qui les domine de plus d'un facteur constant.

Les sections 8.2, 8.3 et 8.4 examinent trois algorithmes, le tri par dénombrement, le tri par base et le tri par paquets qui s'exécutent dans un temps linéaire. Il va de soi que ces algorithmes font appel à des opérations autres que des comparaisons. Le minorant $\Omega(n \lg n)$ ne les concerne donc pas.

8.1 MINORANTS POUR LE TRI

Dans un tri par comparaison, on se sert uniquement de comparaisons d'éléments pour obtenir des informations sur l'ordre d'une séquence d'entrée $\langle a_1, a_2, \dots, a_n \rangle$.

Autrement dit, étant donnés deux éléments a_i et a_j , on effectue l'un des tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ ou $a_i > a_j$ pour déterminer leur ordre relatif.

Dans cette section, on supposera sans nuire à la généralité que tous les éléments d'entrée sont distincts. Grâce à cette hypothèse, les comparaisons de la forme $a_i = a_j$ deviennent inutiles et on peut donc supposer qu'aucune comparaison de ce type n'est effectuée. On remarque également que les comparaisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ et $a_i < a_j$ sont toutes équivalentes, en ce sens où elles fournissent des informations identiques concernant l'ordre relatif de a_i et a_j . On suppose donc que toutes les comparaisons seront de la forme $a_i \leq a_j$.

a) Modèle d'arbre de décision

Les tris par comparaison peuvent être considérés de façon abstraite en termes d'**arbres de décision**. Un arbre de décision est un arbre binaire plein qui représente les comparaisons entre éléments effectuées par un algorithme de tri lorsqu'il traite une entrée d'une taille donnée. Instructions de contrôle, transferts de données et tous les autres aspects de l'algorithme sont ignorés. La figure 8.1 montre l'arbre de décision qui correspond à l'algorithme du tri par insertion (section 2.1) exécuté sur une séquence de trois éléments.

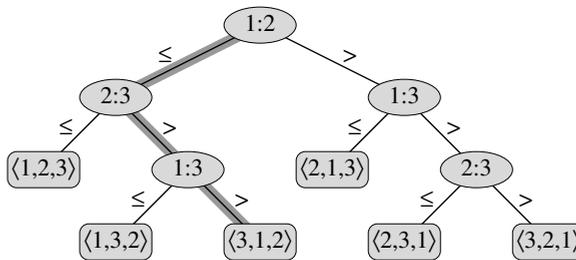


Figure 8.1 Arbre de décision pour un tri par insertion opérant sur trois éléments. Un nœud interne annoté par $i : j$ indique une comparaison entre a_i et a_j . Une feuille annotée par la permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ indique l'ordre $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$. Le chemin ombré indique les décisions faites lors du tri de la séquence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; la permutation $\langle 3, 1, 2 \rangle$ au niveau de la feuille indique que l'ordre trié est $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. Il y a $3! = 6$ permutations possibles pour les éléments en entrée, de sorte que l'arbre de décision doit avoir au moins 6 feuilles.

Dans un arbre de décision, chaque nœud interne est étiqueté par $a_i : a_j$ pour un certain i et un certain j de l'intervalle $1 \leq i, j \leq n$, où n est le nombre d'éléments de la séquence d'entrée. Chaque feuille est étiquetée par une permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$. (Voir section C.1 pour un rappel sur les permutations.) L'exécution de l'algorithme de tri suit un chemin qui part de la racine de l'arbre de décision pour aboutir à une feuille. Sur chaque nœud interne, on effectue une comparaison $a_i \leq a_j$. Les comparaisons suivantes auront lieu dans le sous-arbre gauche si $a_i \leq a_j$ et dans le sous-arbre droit si $a_i > a_j$. Quand on arrive sur une feuille, l'algorithme de tri

a établi l'ordre $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$. Comme tout algorithme de tri correct doit être capable de produire toutes les permutations possibles de son entrée, une condition nécessaire pour qu'un tri par comparaison soit correct est que chacune des $n!$ permutations sur n éléments doit apparaître en tant que l'une des feuilles de l'arbre de décision et que chacune de ces feuilles doit être accessible depuis la racine via un chemin qui corresponde à une exécution concrète du tri par comparaison. (Nous dirons de ces feuilles qu'elles sont « accessibles ».) Nous ne considérerons donc que les arbres de décision dans lesquels chaque permutation figure en tant que feuille accessible.

b) Minorant pour le cas le plus défavorable

La longueur du plus long chemin reliant la racine d'un arbre de décision à l'une quelconque de ses feuilles accessibles représente le nombre de comparaisons effectuées par l'algorithme de tri dans le cas le plus défavorable. Le nombre de comparaisons du cas le plus défavorable est donc égal, pour un algorithme de tri par comparaison donné, à la hauteur de son arbre de décision. Une minorant pour les hauteurs de tous les arbres de décision dans lesquels chaque permutation apparaît en tant que feuille accessible est donc un minorant du temps d'exécution pour n'importe quel algorithme de tri par comparaison. Le théorème suivant calcule un tel minorant.

Théorème 8.1 *Tout algorithme de tri par comparaison exige $\Omega(n \lg n)$ comparaisons dans le cas le plus défavorable.*

Démonstration : La discussion précédente montre qu'il suffit de déterminer la hauteur d'un arbre de décision dans lequel chaque permutation apparaît en tant que feuille accessible. Considérons un arbre de décision de hauteur h avec l feuille accessible correspondant à un tri par comparaison sur n éléments. Comme chacune des $n!$ permutations de l'entrée apparaît sous la forme d'une certaine feuille, on a $n! \leq l$. Puisqu'un arbre binaire de hauteur h n'a pas plus de 2^h feuilles, on a

$$n! \leq l \leq 2^h,$$

ce qui, en prenant les logarithmes, implique

$$\begin{aligned} h &\geq \lg(n!) && \text{(car la fonction } \lg \text{ est monotone croissante)} \\ &= \Omega(n \lg n) && \text{(d'après l'équation (3.18))} . \end{aligned}$$

□

Corollaire 8.2 *Le tri par tas et le tri par fusion sont des tris par comparaison asymptotiquement optimaux.*

Démonstration : Les majorants $O(n \lg n)$ des temps d'exécution du tri par tas et du tri par fusion correspondent au minorant du cas le plus défavorable, minorant qui est $\Omega(n \lg n)$ d'après le théorème 8.1. □

Exercices

8.1.1 Quelle est la plus petite profondeur possible d'une feuille dans un arbre de décision d'un tri par comparaison ?

8.1.2 Obtenir des bornes asymptotiquement serrées pour $\lg(n!)$ sans utiliser l'approximation de Stirling. Évaluer plutôt la sommation $\sum_{k=1}^n \lg k$ à l'aide des techniques vues à la section A.2.

8.1.3 Montrer qu'il n'existe aucun tri par comparaison dont le temps d'exécution soit linéaire pour au moins la moitié des $n!$ entrées possibles de longueur n . Qu'en est-il pour une fraction $1/n$ des entrées de longueur n ? Et pour une fraction $1/2^n$?

8.1.4 Soit à trier une séquence de n éléments. La séquence est constituée de n/k sous-séquences, chacune contenant k éléments. Les éléments d'une sous-séquence donnée sont tous plus petits que les éléments de la sous-séquence suivante et tous plus grands que les éléments de la sous-séquence précédente. Pour trier la séquence complète de longueur n , il suffit donc de trier les k éléments de chacune des n/k sous-séquences. Prouver l'existence d'un minorant $\Omega(n \lg k)$ pour le nombre de comparaisons requis par la résolution de cette variante du problème de tri. (*Conseil* : Se contenter de combiner les minorants des diverses sous-séquences n'est pas une méthode rigoureuse.)

8.2 TRI PAR DÉNOMBREMENT

Le *tri par dénombrement* suppose que chacun des n éléments de l'entrée est un entier de l'intervalle 0 à k , k étant un certain nombre entier. Lorsque $k = O(n)$, le tri s'exécute en un temps $\Theta(n)$.

Le principe du tri par dénombrement est de déterminer, pour chaque élément x de l'entrée, le nombre d'éléments inférieurs à x . Cette information peut servir à placer l'élément x directement à sa position dans le tableau de sortie. Par exemple, s'il existe 17 éléments inférieurs à x , alors x se trouvera en sortie à la position 18. Ce schéma doit être légèrement modifié pour gérer la situation dans laquelle plusieurs éléments ont la même valeur, puisqu'on ne veut pas tous les placer à la même position.

Dans le code du tri par dénombrement, on suppose que l'entrée est un tableau $A[1..n]$ et donc que $\text{longueur}[A] = n$. Nous avons besoin de deux autres tableaux : le tableau $B[1..n]$ contient la sortie triée et le tableau $C[0..k]$ sert d'espace de stockage temporaire.

TRI-DÉNOMBREMENT(A, B, k)

```

1  pour  $i \leftarrow 0$  à  $k$ 
2      faire  $C[i] \leftarrow 0$ 
3  pour  $j \leftarrow 1$  à  $\text{longueur}[A]$ 
4      faire  $C[A[j]] \leftarrow C[A[j]] + 1$ 
5  ▷  $C[i]$  contient maintenant le nombre d'éléments égaux à  $i$ .
6  pour  $i \leftarrow 1$  à  $k$ 
7      faire  $C[i] \leftarrow C[i] + C[i - 1]$ 
8  ▷  $C[i]$  contient maintenant le nombre d'éléments inférieurs ou égaux à  $i$ .
9  pour  $j \leftarrow \text{longueur}[A]$  jusqu'à 1
10     faire  $B[C[A[j]]] \leftarrow A[j]$ 
11      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

La figure 8.2 illustre le tri par dénombrement. Après initialisation de la boucle **pour** des lignes 1–2, on regarde chaque élément de l'entrée dans la boucle **pour** des lignes 3–4. Si la valeur d'un élément est i , on incrémente $C[i]$. Ainsi, après la ligne 4, $C[i]$ contient le nombre d'éléments égaux à i , pour tout entier $i = 0, 1, \dots, k$. Dans les lignes 6–7, on détermine, pour $i = 0, 1, \dots, k$, le nombre d'éléments qui sont inférieurs ou égaux à i et ce en gégrant un cumul constamment actualisé du tableau C .

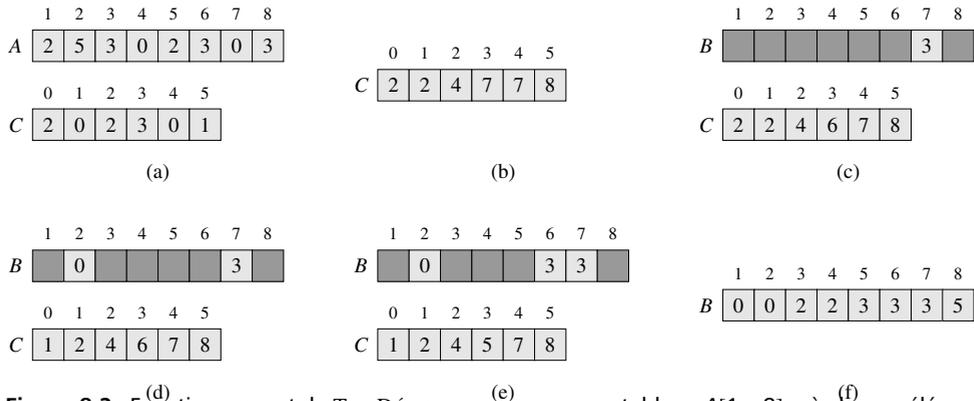


Figure 8.2 Fonctionnement de TRI-DÉNOMBREMENT sur un tableau $A[1..8]$, où chaque élément de A est un entier positif pas plus grand que $k = 5$. (a) Le tableau A et le tableau auxiliaire C après la ligne 4. (b) Le tableau C après la ligne 7. (c)–(e) Le tableau en sortie B et le tableau auxiliaire C après une, deux et trois itérations de la boucle des lignes 9–11. Seules les cases en gris clair du tableau B ont été remplies. (f) Le tableau résultant final B .

Enfin, dans la boucle **pour** des lignes 9–11, on place chaque élément $A[j]$ à sa bonne place dans le tableau de sortie B . Si les n éléments sont tous distincts, alors quand on arrive pour la première fois sur la ligne 9, pour chaque $A[j]$ la valeur $C[A[j]]$ est la position finale correcte de $A[j]$ dans le tableau de sortie, car il y a $C[A[j]]$ éléments inférieurs ou égaux à $A[j]$. Comme les éléments pourraient ne pas être distincts, on décrémente $C[A[j]]$ chaque fois que l'on place une valeur $A[j]$ dans le tableau B . Décrémenter $C[A[j]]$ entraîne que le prochain élément qui a une valeur égale à $A[j]$, s'il y en a un, ira à la position située juste avant $A[j]$ dans le tableau de sortie.

Combien de temps consomme le tri par dénombrement ? La boucle **pour** des lignes 1–2 prend un temps $\Theta(k)$, la boucle **pour** des lignes 3–4 prend un temps $\Theta(n)$, la boucle **pour** des lignes 6–7 prend un temps $\Theta(k)$ et la boucle **pour** des lignes 9–11 prend un temps $\Theta(n)$. Ainsi, le temps global est $\Theta(k + n)$. En pratique, on utilise généralement le tri par dénombrement quand $k = O(n)$, auquel cas le temps d'exécution est $\Theta(n)$.

Le tri par dénombrement améliore le minorant $\Omega(n \lg n)$ établi à la section 8.1 car ce n'est pas un tri par comparaison. En fait, le code ne contient aucune comparaison entre éléments de l'entrée. Le tri par dénombrement utilise à la place les valeurs réelles des éléments pour indexer un tableau. Le minorant $\Omega(n \lg n)$ ne s'applique plus quand on s'éloigne du modèle de tri par comparaison.

Le tri par dénombrement possède une propriété intéressante, à savoir la *stabilité* : les nombres égaux apparaissent dans le tableau de sortie avec l'ordre qu'ils avaient dans le tableau d'entrée. Autrement dit, une égalité éventuelle entre deux nombres est arbitrée par la règle selon laquelle quand un nombre apparaît en premier dans le tableau d'entrée, il apparaît aussi en premier dans le tableau de sortie. En principe, la stabilité n'est importante que si l'élément trié est accompagné de données satellites. Mais la stabilité présente aussi un autre intérêt : le tri par dénombrement sert souvent de sous-routine au tri par base. Comme vous le verrez à la section suivante, la stabilité du tri par dénombrement est un élément clé pour le bon fonctionnement du tri par base.

Exercices

8.2.1 En s'inspirant de la figure 8.2, illustrer l'action de TRI-DÉNOMBREMENT sur le tableau $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2.2 Démontrer que TRI-DÉNOMBREMENT est stable.

8.2.3 Supposez que l'on réécrive ainsi l'en-tête de la boucle **pour** de la ligne 9 de TRI-DÉNOMBREMENT

9 **pour** $j \leftarrow 1$ à longueur[A]

Montrer que l'algorithme fonctionne encore correctement. L'algorithme modifié est-il stable ?

8.2.4 Décrire un algorithme qui, à partir de n entiers donnés appartenant à l'intervalle 0 à k , effectue un pré traitement après lequel il est capable de répondre en temps $O(1)$ à toute question du genre : combien y a-t-il d'entiers parmi les n qui appartiennent à l'intervalle $[a..b]$. Le temps de pré traitement de votre algorithme devra être $\Theta(n+k)$.

8.3 TRI PAR BASE

Le *tri par base* est l'algorithme utilisé par les trieuses de cartes perforées, qu'on ne trouve plus aujourd'hui que dans les musées. Une carte contient 80 colonnes ; dans chaque colonne, on peut faire une perforation à un emplacement choisi parmi 12. La trieuse peut être « programmée » mécaniquement pour examiner une colonne donnée de chaque carte d'un paquet, et distribuer la carte dans un panier parmi 12, selon l'emplacement de la perforation. Un opérateur peut ensuite rassembler les cartes panier par panier, de manière que les cartes perforées à la première position soient au-dessus de celles perforées à la deuxième position et ainsi de suite.

Pour les chiffres décimaux, on n'utilise que 10 positions par colonne. (Les deux autres positions servent à encoder des caractères non numériques.) Un nombre à c

chiffres s'étale donc sur c colonnes. Comme la trieuse de cartes ne peut examiner qu'une seule colonne à la fois, le problème consistant à trier n cartes en fonction d'un numéro à c chiffres requiert un algorithme de tri.

Intuitivement, on pourrait penser à trier les nombres selon le chiffre *le plus significatif*, à trier récursivement chacun des paniers résultants, puis à combiner les paquets dans l'ordre. Malheureusement, comme les cartes de 9 des 10 paniers doivent être mises de côté pour que l'on puisse trier chaque panier, cette procédure oblige à gérer moult piles intermédiaires de cartes. (Voir exercice 8.3.5.)

Le tri par base résout ce problème de manière non intuitive, en commençant par trier en fonction du chiffre *le moins significatif*. Les cartes sont ensuite toutes regroupées, les cartes du panier 0 précédant celles du panier 1, qui elles-mêmes précèdent celles du panier 2, etc. Le paquet tout entier est ensuite trié en fonction du deuxième chiffre le moins significatif, puis reconstitué d'une manière similaire. Le traitement continue jusqu'à ce que les cartes aient été triées sur l'ensemble des c chiffres. Chose remarquable, à ce stade les cartes sont alors complètement triées en fonction des numéros à c chiffres. Il suffit donc de c passes sur le paquet pour qu'il soit trié. La figure 8.3 montre l'action du tri par base sur un « paquet » de sept nombres à 3 chiffres.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 Le fonctionnement du tri par base sur une liste de sept nombres de trois chiffres. La colonne la plus à gauche est l'entrée. Les autres colonnes montrent la liste après des tris successifs, effectués en fonction des différents chiffres (pris dans l'ordre croissant de signification). Les ombrages indiquent le chiffre sur lequel s'est fait le tri qui a produit la liste à partir de la précédente.

Il est essentiel que les tris sur les chiffres soient stables dans cet algorithme. Le tri effectué par une trieuse de cartes perforées est stable, mais l'opérateur doit faire attention à ne pas modifier l'ordre des cartes quand il les sort d'un panier, bien que toutes les cartes d'un panier aient le même chiffre dans la colonne choisie.

Dans un ordinateur classique, qui est une machine séquentielle à accès aléatoire, le tri par base sert parfois à trier des enregistrements de données dont la clé s'étale sur plusieurs champs. Supposons, par exemple, que l'on souhaite trier des dates selon trois clés : année, mois et jour. On pourrait exécuter un algorithme de tri doté d'une fonction de comparaison qui, étant données deux dates, compare les années. Si elles concordent, il compare les mois ; si les mois concordent, il compare les jours. On peut faire la même chose en triant les données trois fois à l'aide d'un tri stable : d'abord selon le jour, puis selon le mois et enfin selon l'année.

Le code du tri par base ne cache aucune subtilité particulière. La procédure suivante suppose que chaque élément du tableau à n éléments A possède c chiffres, le chiffre 1 étant le chiffre d'ordre inférieur et le chiffre c étant le chiffre d'ordre supérieur.

TRI-BASE(A, d)

- 1 **pour** $i \leftarrow 1$ à d
- 2 **faire** employer un tri stable pour trier tableau A selon chiffre i

Lemme 8.3 *Étant donnés n nombres de d chiffres dans lesquels chaque chiffre peut prendre k valeurs possibles, TRI-BASE trie correctement ces nombres en un temps $\Theta(d(n+k))$.*

Démonstration : On établit la validité du tri par base en raisonnant par récurrence sur la colonne en cours de tri (voir exercice 8.3.3). L'analyse du temps d'exécution dépend du tri stable qui sert d'algorithme de tri intermédiaire. Quand chaque chiffre est dans l'intervalle 0 à $k-1$ (il peut donc prendre k valeurs possibles) et quand k n'est pas trop grand, le tri par dénombrement est le choix évident. Chaque passe sur les n nombres à c chiffres prend alors un temps $\Theta(n+k)$. Comme il y a c passes, le temps total du tri par base est $\Theta(d(n+k))$.

Quand d est constant et quand $k = O(n)$, le tri par base s'exécute en temps linéaire. Plus généralement, on dispose d'une certaine souplesse quant à la manière de décomposer chaque clé en chiffres. \square

Lemme 8.4 *Étant donnés n nombres de b bits et un entier positif $r \leq b$, TRI-BASE trie correctement ces nombres en un temps $\Theta((b/r)(n+2^r))$.*

Démonstration : Pour une valeur $r \leq b$, on considère que chaque clé a $d = \lceil b/r \rceil$ chiffres de r bits chacun. Chaque chiffre est un entier appartenant à l'intervalle 0 à $2^r - 1$, de sorte que l'on peut faire du tri par dénombrement en prenant $k = 2^r - 1$. (Par exemple, on peut considérer un mot de 32 bits comme ayant 4 chiffres de 8 bits, de sorte que $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ et $d = b/r = 4$.) Chaque passe du tri par dénombrement prend un temps $\Theta(n+k) = \Theta(n+2^r)$; et il y a d passes, pour un temps d'exécution total de $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$. \square

Pour des valeurs données de n et b , on souhaite prendre la valeur de r , où $r \leq b$, qui minimise l'expression $(b/r)(n+2^r)$. Si $b < \lfloor \lg n \rfloor$, pour toute valeur de $r \leq b$, on a $(n+2^r) = \Theta(n)$. Donc, en choisissant $r = b$, on obtient un temps d'exécution de $(b/b)(n+2^b) = \Theta(n)$, qui est asymptotiquement optimal. Si $b \geq \lfloor \lg n \rfloor$, en choisissant $r = \lfloor \lg n \rfloor$, on obtient le temps optimal à un facteur constant près, ce que l'on peut voir comme suit. Le choix $r = \lfloor \lg n \rfloor$ donne un temps d'exécution $\Theta(bn/\lg n)$. Quand on fait croître r au-dessus de $\lfloor \lg n \rfloor$, le terme 2^r du numérateur augmente plus vite que le terme r du dénominateur; donc, en faisant croître r au-dessus de $\lfloor \lg n \rfloor$, on obtient un temps d'exécution $\Omega(bn/\lg n)$. Si, à la place, on faisait décroître r au-dessous de $\lfloor \lg n \rfloor$, le terme b/r augmenterait et le terme $n+2^r$ resterait à $\Theta(n)$.

Le tri par base est-il préférable à un tri par comparaison comme le tri rapide ? Si $b = O(\lg n)$, comme c'est souvent le cas et que l'on prenne $r \approx \lg n$, alors le temps d'exécution du tri par base est $\Theta(n)$, ce qui semble meilleur que le temps d'exécution moyen du tri rapide qui est $\Theta(n \lg n)$. Les facteurs constants implicites de la notation Θ sont cependant différents. Il se peut que le tri par base fasse moins de passes que le tri rapide sur les n clés, mais chaque passe du tri par base risque de prendre un temps nettement plus long. Le choix de l'algorithme dépendra des caractéristiques des implémentations, de l'ordinateur utilisé (par exemple, le tri rapide utilise souvent les caches matériels plus efficacement que le tri par base) et des données en entrée. En outre, la version du tri par base qui emploie le tri par dénombrement comme tri stable intermédiaire ne trie pas sur place, contrairement à ce que font nombre de tri par comparaison en un temps $\Theta(n \lg n)$. Donc, quand la mémoire principale est une ressource critique, on préférera peut-être prendre un algorithme de tri sur place comme le tri rapide.

Exercices

8.3.1 En s'inspirant de la figure 8.3, illustrer l'action de TRI-BASE sur la liste de mots suivants : BAC, RUE, ROC, MUR, SUD, COQ, DUC, RAT, SAC, MER, TOT, MOU, VER, LAC, EST, BUT.

8.3.2 Parmi les algorithmes de tri suivants, quels sont ceux qui sont stables : tri par insertion, tri par fusion, tri par tas et tri rapide ? Donner un schéma simple qui rende stable n'importe quel algorithme de tri. Combien de temps et d'espace supplémentaires votre schéma demande-t-il ?

8.3.3 Utiliser une récurrence pour prouver que le tri par base fonctionne. À quel endroit de la démonstration a-t-on besoin de supposer que le tri intermédiaire est stable ?

8.3.4 Montrer comment trier n entiers de l'intervalle 0 à $n^2 - 1$ en un temps $O(n)$.

8.3.5 ★ Dans le premier algorithme de tri de cartes donné dans cette section, combien de passes faut-il exactement pour trier des nombres décimaux à c chiffres dans le cas le plus défavorable ? Combien de piles de cartes devrait gérer un opérateur dans le cas le plus défavorable ?

8.4 TRI PAR PAQUETS

Le *tri par paquets* s'exécute en temps linéaire quand l'entrée suit une distribution uniforme. À l'instar du tri par dénombrement, le tri par paquets est rapide car il fait des hypothèses sur l'entrée. Là où le tri par dénombrement suppose que l'entrée se compose d'entiers appartenant à un petit intervalle, le tri par paquets suppose que

l'entrée a été générée par un processus aléatoire qui distribue les éléments de manière uniforme sur l'intervalle $[0, 1)$. (Voir la section C.2 pour la définition d'une distribution uniforme.)

L'idée sous-jacente au tri par paquets est la suivante : on divise l'intervalle $[0, 1)$ en n sous-intervalles de même taille, ou *paquets*, puis on distribue les n nombres de l'entrée dans les différents paquets. Comme les entrées sont distribuées de manière uniforme sur $[0, 1)$, on n'escompte pas qu'un paquet contienne beaucoup de nombres. Pour produire le résultat, on se contente de trier les nombres de chaque paquet, puis de parcourir tous les paquets, dans l'ordre, en énumérant les éléments de chacun.

Notre code du tri par paquets suppose que l'entrée est un tableau à n éléments A et que chaque élément $A[i]$ du tableau satisfait à $0 \leq A[i] < 1$. Le code exige un tableau auxiliaire $B[0..n-1]$ de listes chaînées (paquets) et suppose qu'il existe un mécanisme pour la gestion de ce genre de listes. (La section 10.2 explique comment implémenter les opérations basiques de liste chaînée.)

TRI-PAQUETS(A)

```

1   $n \leftarrow \text{longueur}[A]$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3    faire insérer  $A[i]$  dans liste  $B[\lfloor nA[i] \rfloor]$ 
4  pour  $i \leftarrow 0$  à  $n - 1$ 
5    faire trier liste  $B[i]$  via tri par insertion
6  concaténer les listes  $B[0], B[1], \dots, B[n - 1]$  dans l'ordre
```

La figure 8.4 illustre le fonctionnement du tri par paquets sur un tableau de 10 nombres.

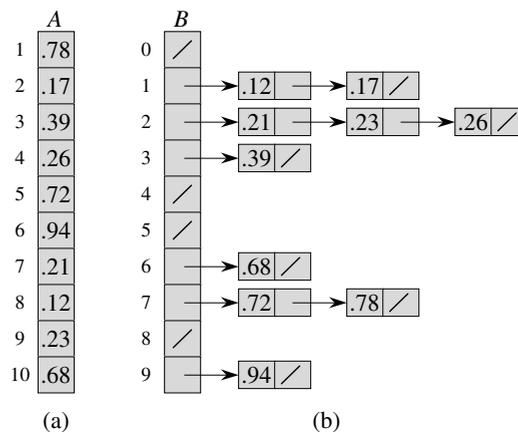


Figure 8.4 Fonctionnement de TRI-PAQUETS. (a) Le tableau en entrée $A[1..10]$. (b) Le tableau $B[0..9]$ de listes (paquets) triées, après la ligne 5 de l'algorithme. Le paquet i contient des valeurs appartenant à l'intervalle semi-ouvert $[i/10, (i+1)/10)$. Le tableau trié consiste en une concaténation ordonnée des listes $B[0], B[1], \dots, B[9]$.

Pour vérifier que cet algorithme est correct, considérons deux éléments $A[i]$ et $A[j]$. On peut supposer, sans nuire à la généralité, que $A[i] \leq A[j]$. Comme $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, l'élément $A[i]$ est placé soit dans le même paquet que $A[j]$, soit dans un paquet d'indice inférieur. Si $A[i]$ et $A[j]$ sont placés dans le même paquet, alors la boucle **pour** des lignes 4–5 les place dans le bon ordre. Si $A[i]$ et $A[j]$ sont placés dans des paquets différents, alors c'est la ligne 6 qui les range dans le bon ordre. Par conséquent, le tri par paquets fonctionne correctement.

Pour analyser le temps d'exécution, observons que toutes les lignes sauf la ligne 5 prennent un temps $O(n)$ dans le cas le plus défavorable. Reste à faire le bilan du temps total consommé par les n appels au tri par insertion en ligne 5.

Pour analyser le coût des appels au tri par insertion, notons n_i la variable aléatoire qui désigne le nombre d'éléments placés dans le paquet $B[i]$. Comme le tri par insertion tourne en temps quadratique (voir section 2.2), le temps d'exécution du tri par paquets est

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Si l'on prend les espérances des deux côtés et que l'on utilise la linéarité de l'espérance, on a

$$\begin{aligned} E[T(n)] &= E \left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E [O(n_i^2)] \quad (\text{d'après la linéarité de l'espérance}) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad (\text{d'après l'équation (C.21)}) . \end{aligned} \quad (8.1)$$

Nous affirmons que

$$E[n_i^2] = 2 - 1/n \quad (8.2)$$

pour $i = 0, 1, \dots, n-1$. Il n'y a rien d'étonnant à ce que chaque paquet i ait la même valeur de $E[n_i^2]$, vu que chaque valeur du tableau d'entrée A a des chances égales de tomber dans l'un quelconque des paquets. Pour prouver l'équation (8.2), on définit des variables indicatrices

$$X_{ij} = I \{A[j] \text{ va dans le paquet } i\}$$

pour $i = 0, 1, \dots, n-1$ et $j = 1, 2, \dots, n$. Par conséquent,

$$n_i = \sum_{j=1}^n X_{ij} .$$

Pour calculer $E[n_i^2]$, on développe le carré puis on regroupe les termes :

$$\begin{aligned}
 E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] \\
 &= E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij}X_{ik}\right] \\
 &= E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} X_{ij}X_{ik}\right] \\
 &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij}X_{ik}] , \tag{8.3}
 \end{aligned}$$

La dernière ligne découle de la linéarité de l'espérance. On évalue les deux sommes séparément. La variable indicatrice X_{ij} est égale à 1 avec la probabilité $1/n$, et égale à 0 sinon ; par conséquent

$$\begin{aligned}
 E[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\
 &= \frac{1}{n} .
 \end{aligned}$$

Quand $k \neq j$, les variables X_{ij} et X_{ik} sont indépendantes ; d'où

$$\begin{aligned}
 E[X_{ij}X_{ik}] &= E[X_{ij}]E[X_{ik}] \\
 &= \frac{1}{n} \cdot \frac{1}{n} \\
 &= \frac{1}{n^2} .
 \end{aligned}$$

En substituant ces deux valeurs attendues dans l'équation (8.3), on obtient

$$\begin{aligned}
 E[n_i^2] &= \sum_{j=1}^n \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2} \\
 &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\
 &= 1 + \frac{n-1}{n} \\
 &= 2 - \frac{1}{n} ,
 \end{aligned}$$

ce qui démontre l'équation (8.2).

En utilisant cette espérance dans l'équation (8.1), on en conclut que le temps attendu du tri par paquet est $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$. Ainsi, l'algorithme complet du tri par paquet s'exécute en temps moyen linéaire.

Même si l'entrée ne provient pas d'une distribution uniforme, le tri par paquet peut encore éventuellement s'exécuter en temps linéaire. Du moment que l'entrée vérifie la propriété selon laquelle la somme des carrés des tailles de paquet est une fonction linéaire du nombre total d'éléments, l'équation (8.1) exprime que le tri par paquet s'exécute en temps linéaire.

Exercices

8.4.1 En s'inspirant de la figure 8.4, illustrer l'action de TRI-PAQUETS sur le tableau $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4.2 Quel est le temps d'exécution, dans le cas le plus défavorable, de l'algorithme du tri par paquet ? Quelle modification simple permettrait à l'algorithme de garder son temps d'exécution moyen linéaire, tout en obtenant un temps d'exécution $O(n \lg n)$ pour le cas le plus défavorable ?

8.4.3 Soit X une variable aléatoire qui est égale au nombre de « pile » dans deux jets d'une pièce non truquée. Combien vaut $E[X^2]$? Et $E^2[X]$?

8.4.4 * Soient n points à l'intérieur du disque unité, $p_i = (x_i, y_i)$, avec $0 < x_i^2 + y_i^2 \leq 1$ pour $i = 1, 2, \dots, n$. On suppose que ces points sont distribués uniformément, c'est à dire que la probabilité de trouver un point dans une région du disque est proportionnelle à la surface de cette région. Concevoir un algorithme ayant un temps moyen $\Theta(n)$ qui puisse trier les n points en fonction de leurs distances $d_i = \sqrt{x_i^2 + y_i^2}$ par rapport à l'origine. (*conseil* : Prendre les tailles des paquets dans TRI-PAQUETS de façon qu'elles reflètent la distribution uniforme des points dans le disque unité.)

8.4.5 * Une *fonction de distribution de probabilité* $P(x)$ d'une variable aléatoire X est définie par $P(x) = \Pr \{X \leq x\}$. Soit une liste de n variables aléatoires X_1, X_2, \dots, X_n provenant d'une fonction P de distribution de probabilité continue qui est calculable en temps $O(1)$. Montrer comment trier ces nombres en temps moyen linéaire.

PROBLÈMES

8.1. Minorants pour le cas moyen d'un tri par comparaison

Dans ce problème, on va démontrer l'existence d'un minorant $\Omega(n \lg n)$ pour le temps d'exécution moyen d'un tri par comparaison quelconque, déterministe ou randomisé, portant sur n entrées distinctes. On commencera par examiner un tri par comparaison déterministe A ayant un arbre de décision T_A . On supposera que toutes les permutations des entrées de A sont équiprobables.

- a. On suppose que chaque feuille de T_A est étiquetée avec la probabilité qu'elle a d'être accessible à partir d'une entrée aléatoire donnée. Démontrer qu'il y a exactement $n!$ feuilles qui sont étiquetées $1/n!$ et que les autres sont étiquetées 0.
- b. Soit $D(T)$ la longueur de chemin extérieur d'un arbre T , c'est à dire le cumul des profondeurs de toutes les feuilles de T . Soit T un arbre de décision à $k > 1$ feuilles, et soient LT et RT les sous-arbres gauche et droite de T . Montrer que $D(T) = D(LT) + D(RT) + k$.
- c. Soit $d(k)$ la valeur minimale de $D(T)$ pour l'ensemble des arbres de décision T à $k > 1$ feuilles. Montrer que $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\}$. (conseil : Considérer un arbre de décision T à k feuilles pour qui le minimum est atteint. Soient i_0 le nombre de feuilles de LT et $k - i_0$ le nombre de feuilles de RT .)
- d. Démontrer que, pour une valeur donnée de $k > 1$ et pour i appartenant à l'intervalle $1 \leq i \leq k - 1$, la fonction $i \lg i + (k - i) \lg(k - i)$ est minimisée en $i = k/2$. En conclure que $d(k) = \Omega(k \lg k)$.
- e. Démontrer que $D(T_A) = \Omega(n! \lg(n!))$ et en conclure que le temps attendu pour trier n éléments est $\Omega(n \lg n)$.

À présent, considérons un tri par comparaison *randomisé* B . On peut étendre le modèle de l'arbre de décision de façon qu'il intègre la randomisation, en y incorporant deux sortes de nœuds : des nœuds de comparaison ordinaires et des nœuds de « randomisation ». Un nœud de randomisation représente un choix aléatoire de la forme $\text{RANDOM}(1, r)$ effectué par l'algorithme B ; le nœud a r enfants, dont chacun a une probabilité égale d'être choisi lors d'une exécution de l'algorithme.

- f. Montrer que, pour un tri par comparaison randomisé B quelconque, il existe un tri par comparaison déterministe A qui n'effectue pas plus de comparaisons que B en moyenne.

8.2. Tri sur place en temps linéaire

Supposez que l'on ait un tableau de n enregistrements de données à trier et que la clé de chaque enregistrement ait la valeur 0 ou 1. Un algorithme de tri pour un tel ensemble d'enregistrements pourrait posséder un certain sous-ensemble des trois caractéristiques souhaitables que voici :

- 1) L'algorithme s'exécute en un temps $O(n)$.
- 2) L'algorithme est stable.
- 3) L'algorithme trie sur place, ne consommant pas plus d'un volume constant d'espace de stockage en plus du tableau original.

- a. Donner un algorithme qui satisfasse aux critères 1 et 2.
- b. Donner un algorithme qui satisfasse aux critères 1 et 3.
- c. Donner un algorithme qui satisfasse aux critères 2 et 3.

- d. L'un quelconque des algorithmes donnés dans les parties (a)–(c) peut-il servir à trier n enregistrements dotés de clés de b bits à l'aide du tri par base en un temps $O(bn)$? Justifier la réponse.
- e. Supposez que les n enregistrements aient des clés appartenant à l'intervalle 1 à k . Montrer comment modifier le tri par dénombrement de façon que les enregistrements soient triés sur place en temps $O(n + k)$. Vous pouvez utiliser un stockage $O(k)$ extérieur au tableau donné en entrée. L'algorithme est-il stable ? (*Conseil* : Comment le feriez-vous pour $k = 3$?)

8.3. Tri d'éléments de longueur variable

- a. Soit un tableau d'entiers dans lequel chaque entier peut avoir un nombre de chiffres différent, sachant que le nombre total de chiffres pour tous les entiers du tableau est n . Montrer comment trier le tableau en temps $O(n)$.
- b. Soit un tableau de chaînes de caractères dans lequel chaque chaîne peut avoir un nombre de caractères différent, sachant que le nombre total de caractères pour l'ensemble des chaînes est n . Montrer comment trier le tableau en temps $O(n)$. (Notez que l'ordre souhaité ici est l'ordre alphabétique standard, par exemple $a < ab < b$.)

8.4. Pots à eau

Vous avez n pots à eau de couleur rouge et n pots à eau de couleur bleue, tous de formes et de tailles différentes. Tous les pots rouges ont des contenances différentes, et c'est aussi le cas des pots bleus. En outre, pour chaque pot rouge, il existe un pot bleu qui a la même contenance, et vice versa.

Votre travail consiste à regrouper les pots par paires, chaque paire étant faite d'un pot rouge et d'un pot bleu de même contenance. Pour ce faire, vous pouvez procéder ainsi : prendre une paire contenant un pot rouge et un pot bleu, remplir le pot rouge d'eau, puis verser l'eau dans le pot bleu. Cette opération vous dira si c'est le pot rouge ou le pot bleu qui peut contenir le plus d'eau, ou bien s'ils ont la même contenance. Supposez qu'une telle comparaison prenne une unité de temps. Vous devez trouver un algorithme qui fasse le minimum de comparaisons pour déterminer le regroupement par paires. Vous ne pouvez pas comparer directement deux pots rouges, ni deux pots bleus.

- a. Décrire un algorithme déterministe qui utilise $\Theta(n^2)$ comparaisons pour regrouper les pots par paires.
- b. Prouver le minorant $\Omega(n \lg n)$ pour le nombre de comparaisons que doit effectuer un algorithme résolvant ce problème.
- c. Donner un algorithme randomisé dont le nombre attendu de comparaisons est $O(n \lg n)$, et prouver la validité de cette borne. Quel est le nombre de comparaisons que fait votre algorithme dans le cas le plus défavorable ?

8.5. Tri par moyenne

Supposez que, au lieu de trier un tableau, on veuille seulement que les éléments augmentent en moyenne. Plus précisément, on dit qu'un tableau à n éléments A est k -trié si, pour tout $i = 1, 2, \dots, n - k$, on a

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$

- Qu'est-ce que cela signifie pour un tableau d'être 1-trié ?
- Donner une permutation des nombres $1, 2, \dots, 10$ qui soit 2-triée, mais pas triée.
- Prouver qu'un tableau à n éléments est k -trié si et seulement si $A[i] \leq A[i+k]$ pour tout $i = 1, 2, \dots, n - k$.
- Donner un algorithme qui k -trie un tableau de n éléments en un temps $O(n \lg(n/k))$.

On peut aussi démontrer un minorant pour le temps requis par la production d'un tableau k -trié, quand k est une constante.

- Montrer qu'un tableau k -trié de longueur n peut être trié en un temps $O(n \lg k)$. (*Conseil* : Utiliser la solution de l'exercice 6.5.8.)
- Montrer que, quand k est une constante, il faut un temps $\Omega(n \lg n)$ pour k -trier un tableau à n éléments. (*Conseil* : Utiliser la solution de la partie précédente, ainsi que le minorant des tris par comparaison.)

8.6. Minorant pour fusion de listes triées

La fusion de deux listes triées est un problème qui revient souvent. C'est une sous-routine de TRI-FUSION et la procédure de fusion de deux listes triées est donnée en tant que FUSION à la section 2.3.1. Dans ce problème, on va montrer qu'il existe un minorant $2n - 1$ pour le nombre de comparaisons requis, dans le cas le plus défavorable, par la fusion de deux listes triées ayant chacune n éléments.

On montrera d'abord un minorant de $2n - o(n)$ comparaisons, en utilisant un arbre de décision.

- Montrer que, étant donnés $2n$ nombres, il y a $\binom{2n}{n}$ façons possibles de les diviser en deux listes triées de n nombres chacune.
- En employant un arbre de décision, montrer que tout algorithme qui fusionne correctement deux listes triées effectue au moins $2n - o(n)$ comparaisons.

On va maintenant exhiber une borne un peu plus fine de $2n - 1$.

- Montrer que, si deux éléments sont consécutifs dans l'ordre trié et qu'ils proviennent de listes opposées, alors ils sont forcément comparés.
- Utiliser la réponse de la partie précédente pour exhiber un minorant de $2n - 1$ comparaisons pour la fusion de deux listes triées.

NOTES

Le modèle d'arbre de décision pour l'étude des tris par comparaison a été introduit par Ford et Johnson [94]. L'exposé très complet Knuth sur le tri [185] présente de nombreuses variantes du problème du tri, dont le minorant théorique sur la complexité du tri donné ici. Des mineurants pour le tri basés sur des généralisations du modèle d'arbre de décision ont été étudiés de manière exhaustive par Ben-Or [36].

Knuth attribue à H. H. Seward l'invention du tri par dénombrement en 1954, ainsi que l'idée de combiner tri par dénombrement et tri par base. Le tri par base, dans lequel on commence par le chiffre le moins significatif, semble être un algorithme traditionnellement utilisé par les opérateurs des trieuses mécaniques de cartes perforées. Selon Knuth, la première référence publiée sur cette méthode est un document de 1929, où L. J. Comrie décrit des équipements pour cartes perforées. Le tri par paquet est utilisé depuis 1956, date où l'idée initiale fut proposée par E. J. Isaac et R. C. Singleton.

Munro et Raman [229] donnent un algorithme de tri stable qui fait $O(n^{1+\varepsilon})$ comparaisons dans le cas le plus défavorable, où $0 < \varepsilon \leq 1$ est une constante quelconque fixée à l'avance. Les algorithmes de tri en temps $O(n \lg n)$ font moins de comparaisons, mais l'algorithme de Munro et Raman ne fait que $O(n)$ transferts de données et il trie sur place.

Le problème du tri de n entiers de b bits en temps $o(n \lg n)$ a été étudié par de nombreux chercheurs. On a obtenu plusieurs résultats positifs, chacun d'eux faisant des hypothèses un peu différentes sur le modèle de calcul et sur les restrictions imposées à l'algorithme. Tous les résultats supposent que la mémoire de l'ordinateur est divisée en mots adressables de b bits. Fredman et Willard [99] ont introduit la structure de données d'arbre de fusion, qu'ils ont utilisée pour trier n entiers en temps $O(n \lg n / \lg \lg n)$. Cette borne a été ultérieurement améliorée en $O(n\sqrt{\lg n})$ par Andersson [16]. Ces algorithmes exigent l'emploi de la multiplication et de plusieurs constantes précalculées. Andersson, Hagerup, Nilsson et Raman [17] ont montré comment trier n entiers en temps $O(n \lg \lg n)$ sans faire appel à la multiplication, mais leur méthode exige de l'espace de stockage qui peut être non borné par rapport à n . Via hachage multiplicatif, l'on peut diminuer l'espace en le faisant tomber à $O(n)$, mais la borne $O(n \lg \lg n)$ du cas le plus défavorable du temps d'exécution devient une borne de temps moyen. En généralisant les arbres de recherche exponentiels de Andersson [16], Thorup [297] a donné un algorithme de tri en temps $O(n(\lg \lg n)^2)$ qui ne fait pas appel à la multiplication, ni à la randomisation, et qui utilise de l'espace linéaire. En combinant ces techniques avec des idées nouvelles, Han [137] a amélioré la borne du tri, la faisant passer à $O(n \lg \lg n \lg \lg \lg n)$. Tous ces algorithmes sont des avancées théoriques majeures, mais ils sont plutôt complexes et, pour l'instant, ils ne semblent pas pouvoir remplacer, en pratique, les algorithmes de tri existants.

Chapitre 9

Médians et rangs

Le i -ème *rang* d'un ensemble de n éléments est le i -ème plus petit élément. Par exemple, le *minimum* d'un ensemble d'éléments est l'élément de rang 1 ($i = 1$), et le *maximum* est l'élément de rang n ($i = n$). Un *médian*, de manière informelle, est le « point du milieu » de l'ensemble. Si n est impair, le médian est unique et a le rang $i = (n + 1)/2$. Si n est pair, il existe deux médians, de rang $i = n/2$ et $i = n/2 + 1$. Donc, si l'on ne tient pas compte de la parité de n , les médians se trouvent aux rangs $i = \lfloor (n + 1)/2 \rfloor$ (*médian inférieur*) et $i = \lceil (n + 1)/2 \rceil$ (*médian supérieur*). Toutefois, pour simplifier l'écriture, pour nous le terme « médian » désignera le médian inférieur.

Ce chapitre va traiter du problème de la sélection du i -ème rang dans un ensemble de n nombres distincts. Nous supposerons, par commodité, que l'ensemble contient des nombres distincts, bien que presque tout ce que nous verrons puisse s'appliquer au cas où un ensemble contient des doublons. Le *problème de la sélection* peut être spécifié formellement de la façon suivante :

Entrée : Un ensemble A de n nombres (distincts) et un nombre i tel que $1 \leq i \leq n$.

Sortie : L'élément $x \in A$ qui est plus grand que $i - 1$ (exactement) autres éléments de A .

Le problème de la sélection peut être résolu en temps $O(n \lg n)$; en effet, on peut trier les nombres à l'aide du tri par tas ou du tri par fusion, puis se contenter d'indexer le i -ème élément du tableau de sortie. Il existe cependant des algorithmes plus rapides.

À la section 9.1, on examinera le problème de la sélection du minimum et du maximum d'un ensemble d'éléments. Le problème général de la sélection est plus

intéressant, et il sera étudié dans les deux sections suivantes. La section 9.2 analyse un algorithme pratique dont le temps d'exécution possède une borne $O(n)$ dans le cas moyen. La section 9.3 propose un algorithme d'intérêt plus théorique dont le temps d'exécution est $O(n)$ dans le cas le plus défavorable.

9.1 MINIMUM ET MAXIMUM

Combien de comparaisons faut-il effectuer pour déterminer le minimum d'un ensemble de n éléments ? On peut facilement obtenir un majorant de $n - 1$ comparaisons : on examine un par un chaque élément de l'ensemble et l'on mémorise le plus petit élément provisoire. Dans la procédure suivante, on suppose que l'ensemble est le tableau A , avec $\text{longueur}[A] = n$.

```

MINIMUM(A)
1   $min \leftarrow A[1]$ 
2  pour  $i \leftarrow 2$  à  $\text{longueur}[A]$ 
3      faire si  $min > A[i]$ 
4          alors  $min \leftarrow A[i]$ 
5  retourner  $min$ 

```

On peut, bien sûr, trouver le maximum avec $n - 1$ comparaisons également.

Est-il possible de faire mieux ? Oui, car on peut obtenir un minorant de $n - 1$ comparaisons pour le problème consistant à déterminer le minimum. On peut imaginer un algorithme qui détermine le minimum sous la forme d'un tournoi. Chaque comparaison est un match, remporté par le plus petit élément de la paire concernée. Il faut bien comprendre ici que tout élément, hormis le vainqueur, perdra au moins un match. Il faudra donc $n - 1$ comparaisons pour déterminer le minimum, et l'algorithme MINIMUM est optimal pour ce qui concerne le nombre de comparaisons effectuées.

a) Minimum et maximum simultanément

Dans certaines applications, on doit trouver en même temps le minimum et le maximum d'un ensemble de n éléments. Par exemple, un programme graphique peut avoir besoin de changer l'échelle d'un ensemble de données (x, y) pour le faire tenir sur un écran rectangulaire ou sur tout autre périphérique de sortie graphique. Pour cela, le programme doit commencer par déterminer le minimum et le maximum de chaque coordonnée.

Il n'est pas très difficile de construire un algorithme capable de trouver à la fois le minimum et le maximum de n éléments avec un nombre de comparaisons $\Theta(n)$, ce qui est asymptotiquement optimal. Il suffit de trouver indépendamment le minimum et le maximum en utilisant $n - 1$ comparaisons pour chacun, ce qui donnera un total de $2n - 2$ comparaisons.

En fait, il suffit de $3 \lfloor n/2 \rfloor$ comparaisons au plus pour trouver à la fois le minimum et le maximum. La stratégie consiste à mémoriser les éléments minimal et maximal provisoires. Au lieu de traiter chaque élément de l'entrée en le comparant avec les minimum et maximum courants, ce qui ferait deux comparaisons par élément, on traite les éléments par paires. On commence par comparer *entre eux* les éléments de chaque paire, puis l'on compare le plus petit des deux avec le minimum provisoire et le plus grand des deux avec le maximum provisoire, ce qui ne fait plus que trois comparaisons par paire d'éléments.

La définition des valeurs initiales des minimum et maximum provisoires varie selon que n est pair ou non. Si n est impair, on prend comme minimum et maximum la valeur du premier élément, puis on traite le reste des éléments par paires. Si n est pair, on fait 1 comparaison sur les deux premiers éléments pour déterminer les valeurs initiales du minimum et du maximum, puis on traite le reste des éléments par paires comme dans le cas n impair.

Analysons le nombre total de comparaisons. Si n est impair, alors on fait $3 \lfloor n/2 \rfloor$ comparaisons. Si n est pair, on fait 1 comparaison initiale, suivie de $3(n-2)/2$ comparaisons, pour un total de $3n/2 - 2$. Donc, dans l'un ou l'autre cas, le nombre total de comparaisons est au plus de $3 \lfloor n/2 \rfloor$.

Exercices

9.1.1 Montrer que le deuxième plus petit élément parmi n peut être trouvé avec $n + \lceil \lg n \rceil - 2$ comparaisons dans le cas le plus défavorable. (*Conseil* : Trouver aussi le plus petit élément.)

9.1.2 * Montrer qu'il faut $\lceil 3n/2 \rceil - 2$ comparaisons, dans le cas le plus défavorable, pour trouver à la fois le maximum et le minimum de n nombres. (*Conseil* : Compter combien il y a de nombres qui sont potentiellement susceptibles d'être le maximum ou le minimum, puis étudier la façon dont une comparaison affecte ces quantités.)

9.2 SÉLECTION EN TEMPS MOYEN LINÉAIRE

Le problème général de la sélection s'avère plus complexe que le simple problème consistant à trouver un minimum bien que, étonnamment, le temps d'exécution asymptotique soit identique : $\Theta(n)$. Dans cette section, nous présenterons un algorithme diviser-pour-régner pour le problème de la sélection. L'algorithme SÉLECTION-RANDOMISÉE s'inspire de l'algorithme de tri rapide du chapitre 7. Comme pour le tri rapide, l'idée est de partitionner récursivement le tableau d'entrée. Mais contrairement au tri rapide, qui traite récursivement les deux côtés de la partition, SÉLECTION-RANDOMISÉE ne s'occupe que d'un seul côté. Cette différence se reflète dans l'analyse : alors que le tri rapide possède un temps d'exécution attendu $\Theta(n \lg n)$, celui de SÉLECTION-RANDOMISÉE est $\Theta(n)$.

SÉLECTION-RANDOMISÉE utilise la procédure PARTITION-RANDOMISÉE de la section 7.3. C'est donc, comme TRI-RAPIDE-RANDOMISÉ, un algorithme randomisé, puisque son comportement est déterminé en partie par le résultat d'un générateur de nombres aléatoires. Le code suivant pour SÉLECTION-RANDOMISÉE fournit le i -ème plus petit élément du tableau $A[p..r]$.

```

SÉLECTION-RANDOMISÉE( $A, p, r, i$ )
1  si  $p = r$ 
2    alors retourner  $A[p]$ 
3   $q \leftarrow$  PARTITION-RANDOMISÉE( $A, p, r$ )
4   $k \leftarrow q - p + 1$ 
5  si  $i = k$            ▷ la valeur du pivot est la réponse
6    alors retourner  $A[q]$ 
7  sinon si  $i < k$ 
8    alors retourner SÉLECTION-RANDOMISÉE( $A, p, q - 1, i$ )
9  sinon retourner SÉLECTION-RANDOMISÉE( $A, q + 1, r, i - k$ )

```

Après l'exécution de PARTITION-RANDOMISÉE en ligne 3 de l'algorithme, le tableau $A[p..r]$ est partitionné en deux sous-tableaux (éventuellement vides) $A[p..q-1]$ et $A[q+1..r]$ tels que chaque élément de $A[p..q-1]$ soit inférieur ou égal à $A[q]$ qui est, lui-même, inférieur à chaque élément de $A[q+1..r]$. Comme dans le tri rapide, on dit de $A[q]$ que c'est l'élément *pivot*. La ligne 4 de l'algorithme calcule le nombre k d'éléments du sous-tableau $A[p..q]$, c'est-à-dire le nombre d'éléments de la partie inférieure de la partition, plus un pour le pivot. La ligne 5 teste ensuite si $A[q]$ est le i -ème plus petit élément. Si tel est le cas, la procédure retourne $A[q]$. Autrement, l'algorithme détermine si le i -ème plus petit élément se trouve dans $A[p..q-1]$ ou dans $A[q+1..r]$. Si $i < k$, c'est que l'élément désiré se trouve dans la région inférieure de la partition ; il est alors sélectionné récursivement dans le sous-tableau (ligne 8). Si $i > k$, c'est que l'élément souhaité se trouve dans la région supérieure de la partition. Comme on connaît déjà k valeurs qui sont inférieures au i -ème plus petit élément de $A[p..r]$, à savoir les éléments de $A[p..q]$, l'élément souhaité est le $(i-k)$ -ème plus petit élément de $A[q+1..r]$, élément qui est sélectionné récursivement en ligne 9. Le code semble autoriser les appels récursifs à des sous-tableaux de 0 élément, mais l'exercice 9.2.1 vous demandera de montrer qu'un tel cas ne peut pas se produire.

Le temps d'exécution de SÉLECTION-RANDOMISÉE, dans le cas le plus défavorable, est $\Theta(n^2)$, même pour trouver le minimum. En effet, si l'on a beaucoup de malchance, il se peut que le partitionnement se fasse systématiquement autour du plus grand élément restant, et le partitionnement prend un temps $\Theta(n)$. Cela dit, l'algorithme fonctionne bien dans le cas moyen ; et comme il est randomisé, il n'y a aucune entrée particulière qui entraîne systématiquement le cas le plus défavorable.

Le temps requis par SÉLECTION-RANDOMISÉE sur un tableau $A[p..r]$ de n éléments est une variable aléatoire que nous noterons $T(n)$. Nous allons maintenant

exhiber un majorant pour $E[T(n)]$. PARTITION-RANDOMISÉE a des chances équiprobables de retourner n'importe quel élément comme étant le pivot. Donc, pour tout k tel que $1 \leq k \leq n$, le sous-tableau $A[p..q]$ a k éléments (tous inférieurs ou égaux au pivot) avec la probabilité $1/n$. Pour $k = 1, 2, \dots, n$, on définit des variables indicatrices X_k par

$$X_k = I \{ \text{le sous-tableau } A[p..q] \text{ a exactement } k \text{ éléments} \} ,$$

On a donc

$$E[X_k] = 1/n . \quad (9.1)$$

Quand on appelle SÉLECTION-RANDOMISÉE et que l'on choisit $A[q]$ comme pivot, on ne sait pas a priori si l'on va avoir toute de suite la bonne réponse, ou si l'on va devoir appeler récursivement la procédure sur le sous-tableau $A[p..q-1]$, ou encore si on va devoir l'appeler récursivement sur le sous-tableau $A[q+1..r]$. Cette décision dépend de l'endroit où le i -ème plus petit élément se trouve par rapport à $A[q]$. Si l'on suppose $T(n)$ monotone croissante, on peut borner le temps requis par l'appel récursif par le temps qu'exige l'appel récursif sur la plus grande entrée possible. Autrement dit, on suppose, pour obtenir un majorant, que le i -ème élément est toujours dans la région de la partition qui contient le plus grand nombre d'éléments. Pour un appel donné de SÉLECTION-RANDOMISÉE, la variable indicatrice X_k a la valeur 1 pour une et une seule valeur de k , et elle vaut 0 pour tous les autres k . Quand $X_k = 1$, les deux sous-tableaux sur lesquels on serait susceptible de faire de la récursivité ont des tailles $k-1$ et $n-k$. D'où la récurrence suivante

$$\begin{aligned} T(n) &\leq \sum_{k=1}^n X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^n (X_k \cdot T(\max(k-1, n-k)) + O(n)) . \end{aligned}$$

En prenant les espérances, on obtient

$$\begin{aligned} E[T(n)] &\leq E \left[\sum_{k=1}^n X_k \cdot T(\max(k-1, n-k)) + O(n) \right] \\ &= \sum_{k=1}^n E[X_k \cdot T(\max(k-1, n-k))] + O(n) \quad (\text{d'après linéarité de l'espérance}) \\ &= \sum_{k=1}^n E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) \quad (\text{d'après équation (C.23)}) \\ &= \sum_{k=1}^n \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \quad (\text{d'après équation (9.1)}) . \end{aligned}$$

Pour pouvoir appliquer l'équation C.23, on s'appuie sur le fait que X_k et $T(\max(k-1, n-k))$ sont des variables aléatoires indépendantes. L'exercice 9.2.2 vous demandera de justifier cette assertion.

Considérons l'expression $\max(k-1, n-k)$. Nous avons

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{si } k > \lceil n/2 \rceil, \\ n-k & \text{si } k \leq \lceil n/2 \rceil. \end{cases}$$

Si n est pair, chaque terme de $T(\lceil n/2 \rceil)$ à $T(n-1)$ apparaît exactement deux fois dans la sommation ; si n est impair, tous ces termes apparaissent deux fois et $T(\lfloor n/2 \rfloor)$ apparaît une fois. On a donc

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} E[T(k)] + O(n).$$

On va résoudre la récurrence par la méthode de substitution. Supposons que $T(n) \leq cn$ pour une certaine constante c qui satisfait à la condition initiale de la récurrence. On suppose que $T(n) = O(1)$ pour n inférieur à une certaine constante ; on exhibera cette constante ultérieurement. On va aussi choisir une constante a telle que la fonction décrite par le terme $O(n)$ ci-dessus (qui décrit la composante non récursive du temps d'exécution de l'algorithme) soit majorée par an pour tout $n > 0$. En utilisant cette hypothèse de récurrence, on obtient

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + an \\ &= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\ &= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\ &\leq \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2 - 2)(n/2 - 1)}{2} \right) + an \\ &= \frac{2c}{n} \left(\frac{n^2 - n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\ &= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\ &= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\ &\leq \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right). \end{aligned}$$

Pour terminer la démonstration, il faut montrer que, pour n suffisamment grand, cette dernière expression est au plus égale à cn ou, ce qui revient au même, que $cn/4 - c/2 - an \geq 0$. Si l'on ajoute $c/2$ aux deux membres et que l'on met n en facteur, on obtient $n(c/4 - a) \geq c/2$. Si nous choisissons la constante c de telle façon que $c/4 - a > 0$, c'est-à-dire si $c > 4a$, alors nous pouvons diviser les deux côtés par $c/4 - a$, ce qui donne

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

Par conséquent, si l'on suppose que $T(n) = O(1)$ pour $n < 2c/(c - 4a)$, on a $T(n) = O(n)$. Nous en concluons que la sélection d'un rang quelconque, en particulier le médian, prend en moyenne un temps linéaire.

Exercices

9.2.1 Montrer que, dans SÉLECTION-RANDOMISÉE, il n'y a jamais d'appel récursif sur un tableau de longueur 0.

9.2.2 Prouver que la variable indicatrice X_k et la valeur $T(\max(k - 1, n - k))$ sont indépendantes.

9.2.3 Écrire une version itérative de SÉLECTION-RANDOMISÉE.

9.2.4 On utilise SÉLECTION-RANDOMISÉE pour sélectionner l'élément minimum du tableau $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Décrire une séquence de partitions qui produise les performances du cas le plus défavorable pour SÉLECTION-RANDOMISÉE.

9.3 SÉLECTION EN TEMPS LINÉAIRE DANS LE CAS LE PLUS DÉFAVORABLE

Nous allons maintenant examiner un algorithme de sélection dont le temps d'exécution est $O(n)$ dans le cas le plus défavorable. À l'instar de SÉLECTION-RANDOMISÉE, l'algorithme SÉLECTION trouve l'élément souhaité en partitionnant récursivement le tableau d'entrée. Toutefois, l'idée sous-jacente à cet algorithme est de *garantir* un bon découpage lorsque le tableau est partitionné. SÉLECTION utilise l'algorithme de partitionnement déterministe PARTITION du tri rapide (voir section 7.1), modifié de façon à prendre comme paramètre d'entrée l'élément autour duquel se fait le partitionnement.

L'algorithme SÉLECTION détermine le i -ème plus petit élément d'un tableau de $n > 1$ éléments en exécutant les étapes suivantes. (Si $n = 1$, SÉLECTION se contente de retourner l'unique valeur d'entrée comme i -ème plus petit élément.)

- 1) On divise les n éléments du tableau en $\lfloor n/5 \rfloor$ groupes de 5 éléments chacun, plus éventuellement le groupe constitué des $n \bmod 5$ éléments restants.
- 2) On trouve le médian de chacun des $\lfloor n/5 \rfloor$ groupes en commençant par trier par insertion les éléments du groupe (il y en a 5 au plus), puis en prenant le médian de la liste triée des éléments du groupe.
- 3) On utilise SÉLECTION de façon récursive pour trouver le médian x des $\lfloor n/5 \rfloor$ médians trouvés à l'étape 2. (S'il y a un nombre pair de médians, de par notre convention x sera le médian inférieur.)
- 4) On partitionne le tableau d'entrée autour du médian des médians x à l'aide de la version modifiée de PARTITION. Soit k le nombre d'éléments de la région inférieure de la partition, augmenté de un ; ainsi, x est le k -ème plus petit élément et il y a $n - k$ éléments dans la région haute de la partition.
- 5) Si $i = k$, alors on retourne x . Sinon, on utilise SÉLECTION récursivement pour trouver le i -ème plus petit élément de la région inférieure si $i < k$, ou le $(i - k)$ -ème plus petit élément de la région supérieure si $i > k$.

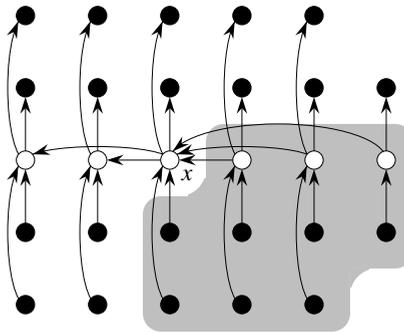


Figure 9.1 Analyse de l'algorithme SÉLECTION. Les n éléments sont représentés par de petits cercles, et chaque groupe occupe une colonne. Les médians des groupes sont coloriés en blanc, et le médian des médians a l'étiquette x . (Pour un nombre pair d'éléments, il s'agit du médian inférieur). Les flèches partent des plus grands éléments vers les plus petits ; on peut voir ainsi que, dans chaque groupe plein (5 éléments) situé à droite de x , il y a 3 éléments qui sont supérieurs à x , et que, dans chaque groupe situé à gauche de x , il y a 3 éléments qui sont inférieurs à x . Les éléments plus grands que x sont sur fond gris.

Pour analyser le temps d'exécution de SÉLECTION, commençons par minorer le nombre d'éléments qui sont supérieurs à l'élément de partitionnement x . La figure 9.1 illustre le processus. La moitié au moins des médians trouvés à l'étape 2 sont supérieurs ⁽¹⁾ au médian des médians x . Donc, la moitié au moins des $\lfloor n/5 \rfloor$ groupes contiennent chacun 3 éléments supérieurs à x , exceptions faites du groupe contenant

(1) Comme nous avons supposé que les nombres étaient distincts, nous pouvons dire « supérieur à » et « inférieur à » sans nous préoccuper d'éventuelles égalités.

moins de 5 éléments (ce groupe n'existe que si 5 ne divise pas n) et du groupe formé de x . Si l'on ne tient pas compte de ces deux groupes, il s'ensuit que le nombre d'éléments supérieurs à x est au moins

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6.$$

De même, le nombre d'éléments inférieurs à x est au moins $3n/10 - 6$. Donc, dans le cas le plus défavorable, SÉLECTION est appelé récursivement sur au plus $7n/10 + 6$ éléments à l'étape 5.

On peut à présent établir une récurrence pour le temps d'exécution $T(n)$ de l'algorithme SÉLECTION dans le cas le plus défavorable. Les étapes 1, 2 et 4 s'exécutent en un temps $O(n)$. (L'étape 2 se compose de $O(n)$ appels au tri par insertion sur des ensembles de taille $O(1)$.) L'étape 3 nécessite un temps $T(\lceil n/5 \rceil)$, et l'étape 5 un temps au plus égal à $T(7n/10 + 6)$, en supposant que T est monotone croissante. Nous ferons l'hypothèse, *a priori* gratuite, que toute entrée de moins de 140 éléments prend un temps $O(1)$; nous verrons bientôt d'où sort cette constante 140. Nous obtenons donc la récurrence

$$T(n) \leq \begin{cases} \Theta(1) & \text{si } n \leq 140, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{si } n > 140. \end{cases}$$

Nous allons montrer, via substitution, que le temps d'exécution est linéaire. Plus précisément, nous allons montrer que $T(n) \leq cn$ pour une constante c choisie suffisamment grande et pour tout $n > 0$. Commençons par supposer que $T(n) \leq cn$ pour une constante c choisie suffisamment grande et pour tout $n \leq 140$; cette hypothèse est vraie dès que c est suffisamment grande. Choisissons également une constante a telle que la fonction décrite par le terme $O(n)$ ci-dessus (qui décrit la composante non récursive du temps d'exécution de l'algorithme) soit majorée par an pour tout $n > 0$. En substituant cette hypothèse de récurrence dans le côté droit de la récurrence, nous obtenons

$$\begin{aligned} T(n) &\leq c \lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an), \end{aligned}$$

qui est au plus égale à cn si

$$-cn/10 + 7c + an \leq 0. \quad (9.2)$$

L'inégalité (9.2) est équivalente à l'inégalité $c \geq 10a(n/(n-70))$ quand $n > 70$. Comme on suppose $n \geq 140$, on a $n/(n-70) \leq 2$; donc, en prenant $c \geq 20a$, on satisfait à l'inégalité (9.2). (Notez qu'il n'y a rien de spécial concernant la constante 140; on pourrait la remplacer par n'importe quel entier strictement supérieur à 70 et choisir ensuite c en conséquence.) Le temps d'exécution du cas le plus défavorable de SÉLECTION est donc linéaire.

Comme dans un tri par comparaison (voir section 8.1), SÉLECTION et SÉLECTION-RANDOMISÉE déterminent des informations sur l'ordre relatif des éléments à l'aide uniquement de comparaisons entre les éléments. Rappelez-vous (chapitre 8) que le tri exige un temps $\Omega(n \lg n)$ dans le modèle de comparaison, même pour le cas moyen (voir problème 8.1). Les algorithmes de tri en temps linéaire vus au chapitre 8 font des hypothèses sur l'entrée. En comparaison, les algorithmes de sélection en temps linéaire vus dans ce chapitre n'exigent pas que l'on fasse des hypothèses sur l'entrée. Ils ne sont pas concernés par le minorant $\Omega(n \lg n)$, car ils arrivent à sélectionner sans faire de tri.

Ainsi, le temps d'exécution est linéaire parce que ces algorithmes ne trient pas ; ce comportement temporel linéaire ne résulte pas d'hypothèses faites sur l'entrée, comme c'était le cas avec les algorithmes de tri du chapitre 8. Le tri exige un temps $\Omega(n \lg n)$ dans le modèle du tri par comparaison, et ce même pour le cas moyen. Par conséquent, la méthode de sélection basée sur le tri et l'indexation, signalée en début de chapitre, est asymptotiquement inefficace.

Exercices

9.3.1 Dans l'algorithme SÉLECTION, les éléments sont répartis par groupes de 5. L'algorithme fonctionnera-t-il en temps linéaire s'ils sont répartis par groupes de 7 ? Montrer que SÉLECTION ne s'exécutera pas en temps linéaire si l'on utilise des groupes de 3 ?

9.3.2 Analyser SÉLECTION pour montrer que, si $n \geq 140$, alors $\lceil n/4 \rceil$ éléments au moins sont supérieurs au médian des médians x et $\lceil n/4 \rceil$ éléments au moins sont inférieurs à x .

9.3.3 Montrer comment améliorer le tri rapide en le faisant s'exécuter en $O(n \lg n)$ dans le cas le plus défavorable.

9.3.4 * On suppose qu'un algorithme utilise uniquement des comparaisons pour trouver le i -ème plus petit élément d'un ensemble à n éléments. Montrer qu'il peut également trouver les $i - 1$ plus petits éléments et les $n - i$ plus grands éléments sans effectuer de comparaison supplémentaire.

9.3.5 Supposez que vous disposiez d'une sous-routine du genre « boîte noire » pour le calcul en temps linéaire du médian dans le cas le plus défavorable. Donner un algorithme simple à temps linéaire qui résolve le problème de la sélection pour un rang arbitraire.

9.3.6 Le k -ième *quantile* d'un ensemble à n éléments est l'ensemble des $k - 1$ rangs qui divisent l'ensemble trié en k sous-ensembles de taille égale (et supérieure à 1). Donner un algorithme en $O(n \lg k)$ pour énumérer le k -ième quantile d'un ensemble.

9.3.7 Décrire un algorithme en $O(n)$ qui, étant donné un ensemble S de n nombres distincts et un entier positif $k \leq n$, détermine les k nombres de S qui sont les plus proches du médian de S .

9.3.8 Soient $X[1..n]$ et $Y[1..n]$ deux tableaux, chacun contenant n nombres déjà triés. Donner un algorithme en $O(\lg n)$ pour trouver le médian des $2n$ éléments présents dans les tableaux X et Y .

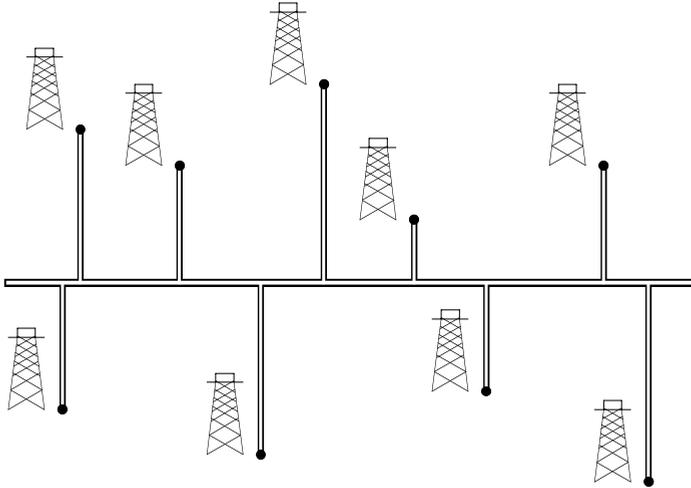


Figure 9.2 On veut trouver, pour l'oléoduc Est-Ouest, un emplacement qui minimise la longueur totale des raccords Nord-Sud.

9.3.9 L'inspecteur Derrick est consultant pour une compagnie pétrolière, qui projette de construire un grand oléoduc d'Est en Ouest à travers un champ pétrolier de n puits. Chaque puit devra être relié à l'oléoduc principal par un chemin minimal (Nord ou Sud), comme le montre la figure 9.2. Connaissant les coordonnées x et y des puits, comment l'inspecteur trouvera-t-il l'emplacement optimal (celui qui minimisera la longueur totale des raccords aux puits) de l'oléoduc principal ? Montrer que cet emplacement peut être déterminé en temps linéaire.

PROBLÈMES

9.1. Les i plus grands nombres en ordre trié

Étant donné un ensemble de n nombres, on souhaite trouver les i plus grands dans l'ordre trié, à l'aide d'un algorithme à base de comparaisons. Trouver l'algorithme qui implémente chacune des méthodes suivantes avec le meilleur temps d'exécution asymptotique pour le cas le plus défavorable, et analyser les temps d'exécution de algorithmes en fonction de n et i .

- Trier les nombres, puis énumérer les i plus grands.
- Construire une file de priorités max à partir des nombres donnés, puis appeler EXTRAIRE-MAX i fois.
- Utiliser un algorithme de sélection pour trouver le i ème plus grand nombre, partitionner autour de ce nombre, puis trier les i plus grands nombres.

9.2. Médian pondéré

Pour n éléments distincts x_1, x_2, \dots, x_n avec des poids positifs p_1, p_2, \dots, p_n tels que $\sum_{i=1}^n p_i = 1$, le **médian pondéré (inférieur)** est l'élément x_k satisfaisant à

$$\sum_{x_i < x_k} p_i < \frac{1}{2}$$

et

$$\sum_{x_i > x_k} p_i \leq \frac{1}{2}.$$

- Démontrer que le médian de x_1, x_2, \dots, x_n est le médian pondéré des x_i affectés des poids $p_i = 1/n$ pour $i = 1, 2, \dots, n$.
- Montrer comment calculer le médian pondéré de n éléments en temps $O(n \lg n)$ dans le cas le plus défavorable, en utilisant un tri.
- Montrer comment calculer le médian pondéré en temps $\Theta(n)$ dans le cas le plus défavorable, à l'aide d'un algorithme à temps linéaire de recherche du médian, tel le SÉLECTION de la section 9.3.

Le **problème de l'emplacement du bureau de poste** est défini comme suit. Soient n points c_1, c_2, \dots, c_n et leurs poids associés p_1, p_2, \dots, p_n . On souhaite trouver un point c (pas nécessairement parmi les points d'entrée) qui minimise la somme $\sum_{i=1}^n p_i d(c, c_i)$, où $d(a, b)$ est la distance entre les points a et b .

- Démontrer que le médian pondéré est une solution optimale pour le problème de l'emplacement du bureau de poste à 1 dimension, dans lequel les points sont simplement des nombres réels et la distance entre les points a et b est $d(a, b) = |a - b|$.
- Trouver la solution optimale pour le problème de l'emplacement du bureau de poste à 2 dimensions, dans lequel les points sont des paires (x, y) de coordonnées et la distance entre les points $a = (x_1, y_1)$ et $b = (x_2, y_2)$ est la **distance de Manhattan** : $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9.3. Petits rangs

On a vu que le nombre $T(n)$ de comparaisons faites par SÉLECTION, dans le cas le plus défavorable, pour sélectionner le i -ème rang parmi n nombres satisfaisait à $T(n) = \Theta(n)$; mais la constante implicite à la notation Θ est plutôt grande. Quand i est petit par rapport à n , on peut implémenter une procédure différente qui utilise

SÉLECTION comme sous-programme mais qui effectue moins de comparaisons dans le cas le plus défavorable.

- a. Décrire un algorithme qui utilise $U_i(n)$ comparaisons pour trouver le i -ème plus petit élément parmi les n , où

$$U_i(n) = \begin{cases} T(n) & \text{si } i \geq n/2, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{sinon.} \end{cases}$$

(Conseil : Commencer par faire $\lfloor n/2 \rfloor$ comparaisons deux à deux disjointes, puis continuer récursivement sur l'ensemble contenant le plus petit élément de chaque paire.)

- b. Montrer que, si $i < n/2$, alors $U_i(n) = n + O(T(2i) \lg(n/i))$.
 c. Montrer que, si i est une constante inférieure à $n/2$, alors $U_i(n) = n + O(\lg n)$.
 d. Montrer que, si $i = n/k$ pour $k \geq 2$, alors $U_i(n) = n + O(T(2n/k) \lg k)$.

NOTES

L'algorithme en temps linéaire de recherche du médian dans le cas le plus défavorable a été inventé par Blum, Floyd, Pratt, Rivest et Tarjan [43]. La version en temps moyen rapide est due à Hoare [146]. Floyd et Rivest [92] ont développé une version améliorée du temps moyen qui partitionne autour d'un élément sélectionné récursivement dans un petit échantillon d'éléments.

On ne sait pas encore exactement combien il faut de comparaisons pour déterminer le médian. Un minorant de $2n$ comparaisons pour la recherche du médian a été exhibé par Bent et John [38]. Un majorant de $3n$ a été donné par Schonhage, Paterson et Pippenger [265]. Dor et Zwick [79] ont amélioré ces deux bornes ; leur majorant est légèrement inférieur à $2,95n$ et leur minorant est légèrement supérieur à $2n$. Paterson [239] décrit ces résultats, ainsi que d'autres travaux afférents.

PARTIE 3

STRUCTURES DE DONNÉES

La notion d'ensemble est aussi fondamentale pour l'informatique que pour les mathématiques. Alors que les ensembles mathématiques sont stables, ceux manipulés par les algorithmes peuvent croître, diminuer, ou subir d'autres modifications au cours du temps. On dit de ces ensembles qu'ils sont *dynamiques*. Les cinq prochains chapitres présentent quelques techniques élémentaires permettant de représenter des ensembles dynamiques finis et de les manipuler sur un ordinateur.

Les types d'opération à effectuer sur les ensembles peuvent varier d'un algorithme à l'autre. Par exemple, de nombreux algorithmes se contentent d'insérer, de supprimer ou de tester l'appartenance. Un ensemble dynamique qui reconnaît ces opérations est appelé *dictionnaire*. D'autres algorithmes nécessitent des opérations plus complexes. Par exemple, les files de priorités min, présentées au chapitre 6 dans le contexte de la structure de données tas, permettent de faire des opérations d'insertion et d'extraction du plus petit élément d'un ensemble. La meilleure façon d'implémenter un ensemble dynamique dépend des opérations qu'il devra reconnaître.

a) *Éléments d'un ensemble dynamique*

Dans une implémentation classique d'un ensemble dynamique, chaque élément est représenté par un objet dont les champs peuvent être examinés et manipulés à l'aide d'un pointeur vers l'objet. (La section 10.3 étudie l'implémentation des objets et des pointeurs dans les environnements de programmation qui ne les proposent pas comme types de données de base.) Certains types d'ensembles dynamiques supposent

que l'un des champs de l'objet contient une *clé* servant d'identifiant. Si les clés sont toutes différentes, on peut considérer l'ensemble dynamique comme un ensemble de valeurs de clés. L'objet peut contenir des *données satellites*, rangées dans d'autres champs de l'objet mais n'intervenant pas dans l'implémentation de l'ensemble. L'objet peut aussi avoir des champs qui sont manipulés par des opérations ensemblistes ; ces champs peuvent contenir des données ou des pointeurs vers d'autres objets de l'ensemble.

Certains ensembles dynamiques présupposent que les clés sont construites à partir d'un ensemble totalement ordonné, comme celui des nombres réels, ou celui de tous les mots classés dans l'ordre alphabétique habituel. (Un ensemble totalement ordonné vérifie la propriété de « trichotomie », définie à la page 47.) Un ordre total permet de définir le plus petit élément d'un ensemble, par exemple, ou bien de parler du prochain élément qui est plus grand qu'un élément donné de l'ensemble.

b) Opérations sur les ensembles dynamiques

Les opérations sur un ensemble dynamique peuvent être regroupées en deux catégories : les *requêtes* qui se contentent de retourner des informations concernant l'ensemble, et les *opérations de modification* qui modifient l'ensemble. Voici une liste des opérations classiques. L'implémentation d'une application particulière ne fera appel en général qu'à une petite partie de ces opérations.

RECHERCHER(S, k) : Une requête qui, étant donné un ensemble S et une valeur de clé k , retourne un pointeur x sur un élément de S tel que $clé[x] = k$, ou NIL si l'élément en question n'appartient pas à S .

INSERTION(S, x) : Une opération de modification qui ajoute à l'ensemble S l'élément pointé par x . On suppose en général que tous les champs de l'élément x requis par la définition de l'ensemble ont déjà été initialisés.

SUPPRESSION(S, x) : Une opération de modification qui, étant donné un pointeur x vers un élément de l'ensemble S , élimine x de S . (Remarquez que cette opération utilise un pointeur vers un élément x , et non une valeur de clé.)

MINIMUM(S) : Une requête sur un ensemble S totalement ordonné qui retourne l'élément de S ayant la plus petite clé.

MAXIMUM(S) : Une requête sur un ensemble S totalement ordonné qui retourne l'élément de S ayant la plus grande clé.

SUCCESEUR(S, x) : Une requête qui, étant donné un élément x dont la clé appartient à un ensemble S totalement ordonné, retourne le prochain élément de S qui est plus grand que x , ou NIL si x est l'élément maximal.

PRÉDÉCESSEUR(S, x) : Une requête qui, étant donné un élément x dont la clé appartient à un ensemble S totalement ordonné, retourne le prochain élément de S qui est plus petit que x , ou NIL si x est l'élément minimal.

Les requêtes SUCCESEUR et PRÉDÉCESSEUR sont souvent étendues à des ensembles ayant des clés non-distinctes. Pour un ensemble à n clés, l'hypothèse habituelle est qu'un appel à MINIMUM suivi de $n - 1$ appels à SUCCESEUR énumère les éléments de l'ensemble dans l'ordre trié.

Le temps nécessaire à l'exécution d'une opération d'ensemble se mesure généralement en fonction de la taille de l'ensemble passé en argument à l'opération. Ainsi, le chapitre 13 décrit une structure de données avec laquelle toutes les opérations susmentionnées se font en $O(\lg n)$ sur un ensemble de taille n .

c) Aperçu de la partie 3

Les chapitres 10–14 présenteront plusieurs structures de données qu'on peut utiliser pour implémenter des ensembles dynamiques ; nombre d'entre elles seront utilisées plus tard pour construire des algorithmes efficaces pour de nombreux problèmes. Une autre structure de données importante, le tas, a déjà été présentée au chapitre 6.

Le chapitre 10 présentera les manipulations essentielles portant sur des structures de données simples comme les piles, les files, les listes chaînées et les arbres enracinés. Il montre également comment implémenter objets et pointeurs dans les environnements de programmation qui ne les proposent pas comme structures de base. Une bonne partie de ces notions sont certainement familières à ceux qui ont suivi des cours de programmation.

Le chapitre 11 présentera les tables de hachage, qui reconnaissent les opérations de dictionnaire INSERTION, SUPPRESSION et RECHERCHER. Dans le pire des cas, le hachage requiert un temps en $\Theta(n)$ pour effectuer une opération RECHERCHER, mais le temps attendu pour les opérations de table de hachage est $O(1)$. L'analyse du hachage fait appel aux probabilités, mais la majeure partie du chapitre ne requiert aucune connaissance préalable sur le sujet.

Le chapitre 12 présentera les arbres de recherche binaires qui reconnaissent toutes les opérations d'ensemble dynamique susmentionnées. Dans le pire des cas, chaque opération prend un temps $\Theta(n)$ pour un arbre à n éléments ; mais sur un arbre construit aléatoirement, le temps attendu pour chaque opération est $O(\lg n)$. Les arbres de recherche binaires servent de base à de nombreuses autres structures de données.

Le chapitre 13 introduira les arbres rouge-noir, variante des arbres de recherche binaires. Contrairement aux arbres de recherche binaires ordinaires, le bon comportement des arbres rouge-noir est garanti : les opérations prennent un temps $O(\lg n)$ dans le pire des cas. Un arbre rouge-noir est un arbre de recherche équilibré ; le chapitre 18 présente un autre type d'arbres de recherche équilibrés, appelés B-arbres. Bien que les arbres rouge-noir soient quelque peu complexes, on pourra se contenter de glaner leurs propriétés fondamentales sans être obligé d'étudier le détail de leurs mécanismes. Par ailleurs, un survol du pseudo code pourra se révéler très instructif.

Le chapitre 14 montrera comment étendre les arbres rouge-noir pour leur ajouter d'autres opérations que les opérations de base précédemment énumérées. Nous verrons d'abord comment gérer dynamiquement les rangs sur un ensemble de clés ; nous verrons ensuite comment gérer des intervalles de nombres réels.

Chapitre 10

Structures de données élémentaires

Dans ce chapitre, nous étudierons la représentation d'ensembles dynamiques à l'aide de structures de données simples utilisant des pointeurs. Bien qu'il soit possible de construire nombre de structures de données complexes grâce aux pointeurs, nous ne présenterons ici que les plus simples : piles, files, listes chaînées et arbres enracinés. Nous étudierons également une méthode permettant de construire des objets et des pointeurs à partir de tableaux.

10.1 PILES ET FILES

Les piles et les files sont des ensembles dynamiques pour lesquels l'élément à supprimer *via* l'opération SUPPRIMER est défini par la nature intrinsèque de l'ensemble. Dans une *pile*, l'élément supprimé est le dernier inséré : la pile met en œuvre le principe *dernier entré, premier sorti*, ou **LIFO** (Last-In, First-Out). De même, dans une *file*, l'élément supprimé est toujours le plus ancien ; la file met en œuvre le principe *premier entré, premier sorti*, ou **FIFO** (First-In, First-Out). Il existe plusieurs manières efficaces d'implémenter des piles et des files dans un ordinateur. Dans cette section, nous allons montrer comment les implémenter à l'aide d'un tableau simple.

a) Piles

L'opération INSÉRER dans une pile est souvent appelée EMPILER et l'opération SUPPRIMER, qui ne prend pas d'élément pour argument, est souvent appelée DÉPILER. Ces noms font allusion aux piles rencontrées dans la vie de tous les jours, comme les piles d'assiettes automatiques en usage dans les cafétérias. L'ordre dans lequel les assiettes sont dépilées est l'inverse de celui dans lequel elles ont été empilées, puisque seule l'assiette supérieure est accessible.

Comme on le voit à la figure 10.1, il est possible d'implémenter une pile d'au plus n éléments avec un tableau $P[1..n]$. Le tableau possède un attribut $sommet[P]$ qui indexe l'élément le plus récemment inséré. La pile est constituée des éléments $P[1..sommet[P]]$, où $P[1]$ est l'élément situé à la base de la pile et $P[sommet[P]]$ est l'élément situé au sommet.

Quand $sommet[P] = 0$, la pile ne contient aucun élément ; elle est *vide*. On peut tester si la pile est vide à l'aide de l'opération de requête PILE-VIDE. Si l'on tente de dépiler une pile vide, on dit qu'elle *débord* négativement, ce qui est en général une erreur. Si $sommet[P]$ dépasse n , on dit que la pile *débord*. (Dans notre pseudo code, on ne se préoccupera pas d'un débordement éventuel de la pile.)

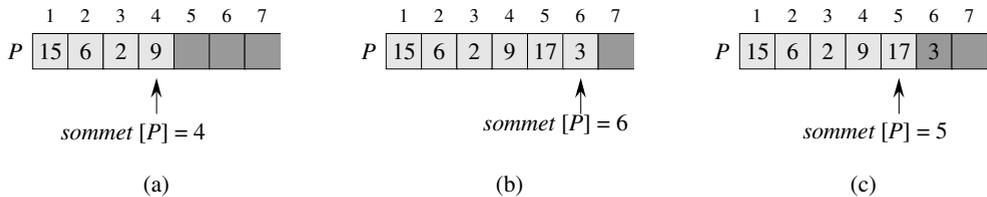


Figure 10.1 Implémentation via un tableau d'une pile P . Les éléments de la pile apparaissent uniquement aux positions en gris clair. (a) La pile P contient 4 éléments. L'élément sommet est 9. (b) L'état de la pile P après les appels $EMPILER(P, 17)$ et $EMPILER(P, 3)$. (c) L'état de la pile P après que l'appel $DÉPILER(P)$ a retourné 3, qui est l'élément le plus récemment empilé. Bien que l'élément 3 apparaisse encore dans le tableau, il n'est plus dans la pile ; le sommet est occupé par l'élément 17.

On peut implémenter chaque opération de pile avec quelques lignes de code.

PILE-VIDE(P)

```

1  si  $sommet[P] = 0$ 
2    alors retourner VRAI
3    sinon retourner FAUX

```

EMPILER(P, x)

```

1   $sommet[P] \leftarrow sommet[P] + 1$ 
2   $P[sommet[P]] \leftarrow x$ 

```

DÉPILER(P)

```

1  si PILE-VIDE( $P$ )
2    alors erreur « débordement négatif »
3    sinon  $sommet[P] \leftarrow sommet[P] - 1$ 
4    retourner  $P[sommet[P] + 1]$ 

```

La figure 10.1 montre l'effet des opérations de modification EMPILER et DÉPILER. Chacune des trois opérations de pile consomme un temps $O(1)$.

b) Files

On appelle ENFILER l'opération INSÉRER sur une file et on appelle DÉFILER l'opération SUPPRIMER ; à l'instar de l'opération de pile DÉPILER, DÉFILER ne prend pas d'argument. La propriété FIFO d'une file la fait agir comme une file à un guichet d'inscription. La file comporte une *tête* et une *queue*. Lorsqu'un élément est enfilé, il prend place à la queue de la file, comme l'étudiant nouvellement arrivé prend sa place à la fin de la file d'inscription. L'élément défilé est toujours le premier en tête de la file, de même que l'étudiant qui est servi au guichet est celui qui a attendu le plus longtemps dans la file. (Heureusement, nous n'avons pas ici à prendre en compte les éléments qui resquillent).

La figure 10.2 montre une manière d'implémenter une file d'au plus $n - 1$ éléments à l'aide d'un tableau $F[1..n]$. La file comporte un attribut $tête[F]$ qui indexe, ou pointe vers, sa tête. L'attribut $queue[F]$ indexe le prochain emplacement où sera inséré un élément nouveau. Les éléments de la file se trouvent aux emplacements $tête[F], tête[F] + 1, \dots, queue[F] - 1$, après quoi l'on « boucle » : l'emplacement 1 suit immédiatement l'emplacement n dans un ordre circulaire. Quand $tête[F] = queue[F]$, la file est vide. Au départ, on a $tête[F] = queue[F] = 1$. Quand la file est vide, tenter de défiler un élément provoque un débordement négatif de la file. Quand $tête[F] = queue[F] + 1$, la file est pleine ; tenter d'enfiler un élément provoque alors un débordement.

Dans nos procédures ENFILER et DÉFILER, le test d'erreur pour les débordements a été omis. (L'exercice 10.1.4 vous demande un code capable de tester les deux causes d'erreurs.)

```

ENFILER( $F, x$ )
1   $F[queue[F]] \leftarrow x$ 
2  si  $queue[F] = longueur[F]$ 
3    alors  $queue[F] \leftarrow 1$ 
4    sinon  $queue[F] \leftarrow queue[F] + 1$ 

DÉFILER( $F$ )
1   $x \leftarrow F[tête[F]]$ 
2  si  $tête[F] = longueur[F]$ 
3    alors  $tête[F] \leftarrow 1$ 
4    sinon  $tête[F] \leftarrow tête[F] + 1$ 
5  retourner  $x$ 

```

La figure 10.2 montre l'effet des opérations ENFILER et DÉFILER. Chaque opération s'effectue en $O(1)$.

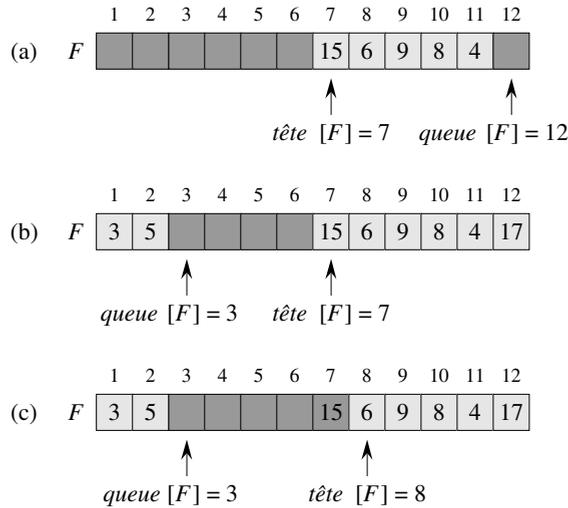


Figure 10.2 Une file implémentée à l'aide d'un tableau $F[1..12]$. Les éléments de la file apparaissent uniquement aux positions en gris clair. (a) La file contient 5 éléments, aux emplacements $F[7..11]$. (b) La configuration de la file après les appels $\text{ENFILER}(F, 17)$, $\text{ENFILER}(F, 3)$ et $\text{ENFILER}(F, 5)$. (c) La configuration de la file après l'appel $\text{DÉFILER}(F)$ qui retourne la valeur de clé 15 précédemment en tête de file. La nouvelle tête à la clé 6.

Exercices

10.1.1 En s'inspirant de la figure 10.1, illustrer le résultat de chacune des opérations $\text{EMPILER}(P, 4)$, $\text{EMPILER}(P, 1)$, $\text{EMPILER}(P, 3)$, $\text{DÉPILER}(P)$, $\text{EMPILER}(P, 8)$ et $\text{DÉPILER}(P)$ sur une pile P , initialement vide, stockée dans le tableau $P[1..6]$.

10.1.2 Expliquer comment implémenter deux piles dans un seul tableau $A[1..n]$ de telle manière qu'aucune pile ne déborde à moins que le nombre total d'éléments dans les deux piles vaille n . Les opérations EMPILER et DÉPILER devront s'exécuter dans un temps $O(1)$.

10.1.3 En s'inspirant de la figure 10.2, illustrer le résultat de chacune des opérations $\text{ENFILER}(F, 4)$, $\text{ENFILER}(F, 1)$, $\text{ENFILER}(F, 3)$, $\text{DÉFILER}(F)$, $\text{ENFILER}(F, 8)$ et $\text{DÉFILER}(F)$ sur une file F , initialement vide, stockée dans le tableau $F[1..6]$.

10.1.4 Réécrire ENFILER et DÉFILER pour détecter les débordements (normaux et négatifs) de file.

10.1.5 Alors qu'une pile n'autorise l'insertion et la suppression des éléments qu'à une seule extrémité et qu'une file autorise l'insertion à une extrémité et la suppression à l'autre extrémité, une *file à double entrée* autorise l'insertion et la suppression à chaque bout. Écrire quatre procédures en $O(1)$ pour insérer et supprimer des éléments de chaque côté d'une file à double entrée construite à partir d'un tableau.

10.1.6 Montrer comment implémenter une file à l'aide de deux piles. Analyser le temps d'exécution des opérations de file.

10.1.7 Montrer comment implémenter une pile à l'aide de deux files. Analyser le temps d'exécution des opérations de pile.

10.2 LISTES CHAÎNÉES

Une *liste chaînée* est une structure de données dans laquelle les objets sont arrangés linéairement. Toutefois, contrairement au tableau, pour lequel l'ordre linéaire est déterminé par les indices, l'ordre d'une liste chaînée est déterminé par un pointeur dans chaque objet. Les listes chaînées fournissent une représentation simple et souple pour les ensembles dynamiques, supportant (pas toujours très efficacement) toutes les opérations énumérées à la page 192.

Comme le montre la figure 10.3, chaque élément d'une *liste doublement chaînée* L est un objet comportant un champ *clé* et deux autres champs pointeurs : *succ* et *préd*. L'objet peut aussi contenir d'autres données satellites. Étant donné un élément x de la liste, $succ[x]$ pointe sur son successeur dans la liste chaînée et $préd[x]$ pointe sur son prédécesseur. Si $préd[x] = \text{NIL}$, l'élément x n'a pas de prédécesseur et est donc le premier élément, aussi appelé *tête* de liste. Si $succ[x] = \text{NIL}$, l'élément x n'a pas de successeur et est donc le dernier élément, aussi appelé *queue* de liste. Un attribut $tête[L]$ pointe sur le premier élément de la liste. Si $tête[L] = \text{NIL}$, la liste est vide.

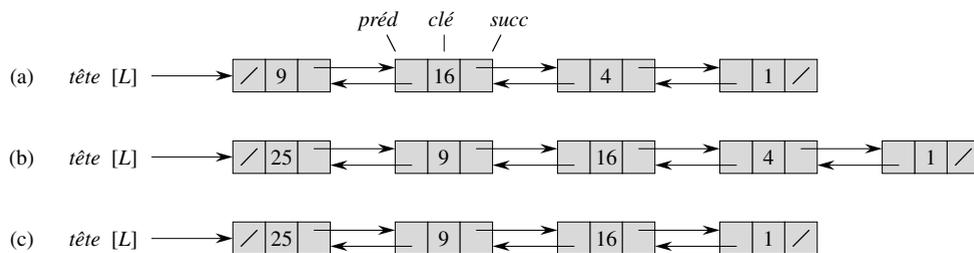


Figure 10.3 (a) Une liste doublement chaînée L représentant l'ensemble dynamique $\{1, 4, 9, 16\}$. Chaque élément de la liste est un objet avec des champs contenant la clé et des pointeurs (représentés par des flèches) sur les objets suivant et précédent. Le champ *succ* de la queue et le champ *préd* de la tête valent NIL, représenté par un slash. L'attribut $tête[L]$ pointe sur la tête. (b) Après l'exécution de `LISTE-INSÉRER(L, x)`, où $clé[x] = 25$, la liste chaînée contient à sa tête un nouvel objet ayant pour clé 25. Ce nouvel objet pointe sur l'ancienne tête de clé 9. (c) Le résultat de l'appel `LISTE-SUPPRIMER(L, x)` ultérieur, où x pointe sur l'objet ayant pour clé 4.

Une liste peut prendre différentes formes. Elle peut être chaînée, ou doublement chaînée, triée ou non, circulaire ou non. Si une liste chaînée est *simple*, on omet le pointeur *préd* de chaque élément. Si une liste est *triée*, l'ordre linéaire de la liste correspond à l'ordre linéaire des clés stockées dans les éléments de la liste ; l'élément minimum est la tête de la liste et l'élément maximum est la queue. Si la liste est *non-triée*, les éléments peuvent apparaître dans n'importe quel ordre. Dans une *liste circulaire*, le pointeur *préd* de la tête de liste pointe sur la queue et le pointeur *succ*

de la queue de liste pointe sur la tête. La liste peut donc être vue comme un anneau d'éléments. Dans le reste de cette section, on suppose que les listes sur lesquelles nous travaillons sont non triées et doublement chaînées.

a) Recherche dans une liste chaînée

La procédure RECHERCHE-LISTE(L, k) trouve le premier élément de clé k dans la liste L par une simple recherche linéaire et retourne un pointeur sur cet élément. Si aucun objet de clé k n'apparaît dans la liste, la procédure retourne NIL. Si l'on prend la liste chaînée de la figure 10.3(a), l'appel RECHERCHE-LISTE($L, 4$) retourne un pointeur sur le troisième élément et l'appel RECHERCHE-LISTE($L, 7$) retourne NIL.

RECHERCHE-LISTE(L, k)

```

1   $x \leftarrow \text{tête}[L]$ 
2  tant que  $x \neq \text{NIL}$  et  $\text{clé}[x] \neq k$ 
3      faire  $x \leftarrow \text{succ}[x]$ 
4  retourner  $x$ 
```

Pour parcourir une liste de n objets, la procédure RECHERCHE-LISTE s'exécute en $\Theta(n)$ dans le cas le plus défavorable, puisqu'on peut être obligé de parcourir la liste entière.

b) Insertion dans une liste chaînée

Étant donné un élément x dont le champ *clé* a déjà été initialisé, la procédure INSÉRER-LISTE « greffe » x à l'avant de la liste chaînée, comme on le voit sur la figure 10.3(b).

INSÉRER-LISTE(L, x)

```

1   $\text{succ}[x] \leftarrow \text{tête}[L]$ 
2  si  $\text{tête}[L] \neq \text{NIL}$ 
3      alors  $\text{préd}[\text{tête}[L]] \leftarrow x$ 
4   $\text{tête}[L] \leftarrow x$ 
5   $\text{préd}[x] \leftarrow \text{NIL}$ 
```

Le temps d'exécution de INSÉRER-LISTE sur une liste de n éléments est $O(1)$.

c) Suppression dans une liste chaînée

La procédure SUPPRIMER-LISTE élimine un élément x d'une liste chaînée L . Il faut lui fournir un pointeur sur x et elle se charge alors de « détacher » x de la liste en mettant les pointeurs à jour. Si l'on souhaite supprimer un élément ayant une clé donnée, on doit commencer par appeler RECHERCHE-LISTE pour récupérer un pointeur sur l'élément.

SUPPRIMER-LISTE(L, x)

```

1  si  $préd[x] \neq \text{NIL}$ 
2    alors  $succ[préd[x]] \leftarrow succ[x]$ 
3    sinon  $tête[L] \leftarrow succ[x]$ 
4  si  $succ[x] \neq \text{NIL}$ 
5    alors  $préd[succ[x]] \leftarrow préd[x]$ 

```

La figure 10.3(c) montre comment un élément est supprimé dans une liste chaînée. SUPPRIMER-LISTE s'exécute dans un temps en $O(1)$; mais si l'on souhaite supprimer un élément à partir de sa clé, il faut un temps $\Theta(n)$ dans le cas le plus défavorable, car on doit commencer par appeler RECHERCHE-LISTE.

d) Sentinelles

Le code de SUPPRIMER-LISTE serait plus simple si l'on pouvait ignorer les conditions aux limites en tête et en queue de liste.

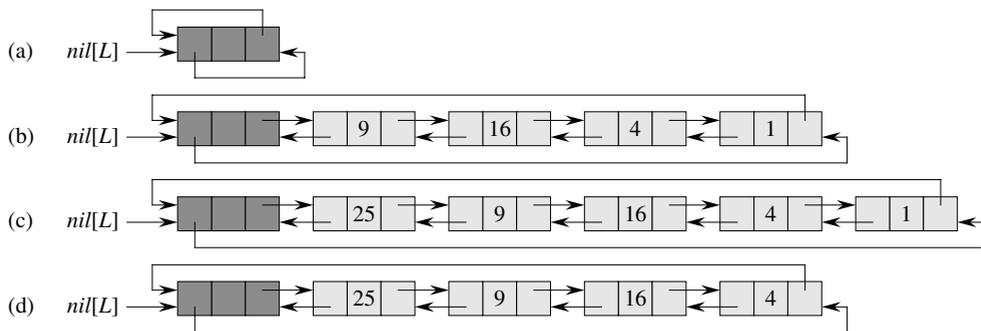


Figure 10.4 Liste circulaire doublement chaînée, avec sentinelle. La sentinelle $nil[L]$ apparaît entre la tête et la queue. L'attribut $tête[L]$ n'est plus nécessaire, car on peut accéder à la tête de la liste via $succ[nil[L]]$. (a) Une liste vide. (b) La liste chaînée de la figure 10.3(a), avec la clé 9 en tête et la clé 1 en queue. (c) La liste après exécution de INSÉRER-LISTE'(L, x), où $clé[x] = 25$. Le nouvel objet devient la tête de la liste. (d) La liste après suppression de l'objet ayant la clé 1. La nouvelle queue est l'objet ayant la clé 4.

SUPPRIMER-LISTE'(L, x)

```

1   $succ[préd[x]] \leftarrow succ[x]$ 
2   $préd[succ[x]] \leftarrow préd[x]$ 

```

Une *sentinelle* est un objet fictif permettant de simplifier les conditions aux limites. Par exemple, supposons qu'on fournisse avec une liste L un objet $nil[L]$ qui représente NIL mais contienne tous les champs des autres éléments de la liste. Partout où nous avons une référence à NIL dans le code d'implémentation de la liste, on le remplace par une référence à la sentinelle $nil[L]$. Comme le montre la figure 10.4, cela transforme une liste doublement chaînée ordinaire en une liste circulaire doublement

chaînée, avec la sentinelle placée entre la tête et la queue ; le champ $succ[nil[L]]$ pointe sur la tête de la liste, et $préd[nil[L]]$ pointe sur la queue. De même, le champ $succ$ de la queue et le champ $préd$ de la tête pointent tous les deux vers $nil[L]$. Comme $succ[nil[L]]$ pointe sur la tête, on peut complètement éliminer l'attribut $tête[L]$, en remplaçant ses références par des références à $succ[nil[L]]$. Une liste vide ne contient que la sentinelle, puisque $succ[nil[L]]$ et $préd[nil[L]]$ peuvent tous deux être initialisés à $nil[L]$.

Le code de RECHERCHE-LISTE reste identique au précédent, mais inclut les modifications des références à NIL et à $tête[L]$ proposées ci-dessus.

```
RECHERCHE-LISTE'(L, k)
1  x ← succ[nil[L]]
2  tant que x ≠ nil[L] et clé[x] ≠ k
3    faire x ← succ[x]
4  retourner x
```

On utilise la procédure de deux lignes SUPPRIMER-LISTE' pour supprimer un élément de la liste. On utilise la procédure suivante pour insérer un élément dans la liste.

```
INSÉRER-LISTE'(L, x)
1  succ[x] ← succ[nil[L]]
2  préd[succ[nil[L]]] ← x
3  succ[nil[L]] ← x
4  préd[x] ← nil[L]
```

La figure 10.4 montre l'effet de INSÉRER-LISTE' et SUPPRIMER-LISTE' sur une liste témoin.

Les sentinelles réduisent rarement les bornes asymptotiques des opérations sur les structures de données, mais elles réduisent parfois les facteurs constants. L'avantage des sentinelles à l'intérieur des boucles est en général un problème de clarté du code plutôt que de vitesse ; le code de la liste chaînée, par exemple, est simplifié par l'utilisation de sentinelles, mais n'économise qu'un temps $O(1)$ dans les procédures INSÉRER-LISTE' et SUPPRIMER-LISTE'. Cela dit, dans d'autres situations, l'utilisation des sentinelles aide à compacter le code à l'intérieur d'une boucle, réduisant ainsi le coefficient de, disons n ou n^2 , dans le temps d'exécution.

On ne devra pas faire un usage inconsidéré des sentinelles. Si l'on travaille sur beaucoup de petites listes, l'espace supplémentaire nécessaire pour stocker leur sentinelles peut représenter une dépense de mémoire significative. Dans ce livre, on n'utilisera les sentinelles que lorsqu'elles simplifient réellement le code.

Exercices

10.2.1 Peut-on implémenter l'opération d'ensemble dynamique INSÉRER sur une liste simplement chaînée pour qu'elle s'exécute dans un temps en $O(1)$? Et pour SUPPRIMER ?

10.2.2 Implémenter une pile à l'aide d'une liste simplement chaînée L . Les opérations EMPIILER et DÉPIILER devront encore s'exécuter en $O(1)$.

10.2.3 Implémenter une file à l'aide d'une liste simplement chaînée L . Les opérations ENFIILER et DÉFIILER devront encore s'exécuter en $O(1)$.

10.2.4 Au niveau de l'écriture, chaque itération de boucle dans la procédure RECHERCHE-LISTE' exige deux tests : un pour $x \neq nil[L]$ et un pour $clé[x] \neq k$. Montrer comment éliminer le test $x \neq nil[L]$ dans chaque itération.

10.2.5 Implémenter les opérations de dictionnaire INSÉRER, SUPPRIMER et RECHERCHER à l'aide de listes circulaires simplement chaînées. Quels sont les temps d'exécution de vos procédures ?

10.2.6 L'opération UNION sur les ensembles dynamiques prend deux ensembles disjoints S_1 et S_2 en entrée et retourne un ensemble $S = S_1 \cup S_2$ constitué de tous les éléments de S_1 et S_2 . Les ensembles S_1 et S_2 sont en général détruits par l'opération. Montrer comment implémenter UNION pour qu'elle s'exécute en $O(1)$, en utilisant une structure de données liste adaptée.

10.2.7 Donner une procédure non récursive en $\Theta(n)$ qui inverse l'ordre d'une liste simplement chaînée à n éléments. En dehors de l'espace nécessaire pour contenir la liste elle-même, la procédure ne devra pas utiliser d'espace de stockage non constant.

10.2.8 * Expliquer comment implémenter des listes doublement chaînées à l'aide d'une seule valeur de pointeur $sp[x]$ par élément au lieu des deux habituelles ($succ$ et $préd$). On suppose que toutes les valeurs de pointeurs peuvent être interprétées comme des entiers sur k bits et on définit $sp[x]$ par $sp[x] = succ[x] \text{ XOR } préd[x]$, le « ou exclusif » sur k bits de $succ[x]$ et $préd[x]$. (La valeur NIL est représentée par 0.) Ne pas oublier de décrire toutes les informations utiles pour accéder à la tête de liste. Montrer comment implémenter les opérations RECHERCHER, INSÉRER et SUPPRIMER sur une telle liste. Montrer également comment on peut inverser une telle liste dans un temps $O(1)$.

10.3 IMPLÉMENTATION DES POINTEURS ET DES OBJETS

Comment des pointeurs et des objets peuvent-ils être implémentés dans des langages comme Fortran, qui ne les proposent pas en standard ? Dans cette section, nous verrons deux manières d'implémenter des structures de données chaînées sans faire appel à un type de données pointeur explicite. Nous créerons les objets et les pointeurs à partir de tableaux et d'indices de tableau.

a) Une représentation des objets par tableaux multiples

On peut représenter une collection d'objets ayant les mêmes champs en utilisant un tableau pour chaque champ. Par exemple, la figure 10.5 montre comment implémenter la liste chaînée de la figure 10.3(a) avec trois tableaux. Le tableau *clé* contient les valeurs des clés qui se trouvent dans l'ensemble dynamique et les pointeurs sont stockés dans les tableaux *succ* et *préd*. Pour un indice de tableau x donné, $clé[x]$, $succ[x]$ et $préd[x]$ représentent un objet de la liste chaînée. Selon cette interprétation, un pointeur x est tout simplement un indice commun aux tableaux *clé*, *succ* et *préd*.

Dans la figure 10.3(a), l'objet de clé 4 suit l'objet de clé 16 dans la liste chaînée. Dans la figure 10.5, la clé 4 apparaît dans $clé[2]$ et la clé 16 apparaît dans $clé[5]$. On a donc $succ[5] = 2$ et $préd[2] = 5$. Bien que la constante NIL apparaisse dans le champ *succ* de la queue et le champ *préd* de la tête, on utilise habituellement un entier (comme 0 ou -1) qui ne peut pas représenter d'indice réel dans les tableaux. Une variable L contient l'indice de la tête de liste.

Dans notre pseudo code, nous avons utilisé des crochets pour représenter à la fois l'indexation d'un tableau et la sélection d'un champ (attribut) d'un objet. Dans l'un ou l'autre cas, la signification de $clé[x]$, $succ[x]$ et $préd[x]$ est cohérente avec les pratiques d'implémentation.

b) Représentation des objets par un tableau unique

Les mots d'une mémoire d'ordinateur sont généralement adressés par des entiers compris entre 0 et $M - 1$, où M est un entier suffisamment grand. Dans nombre de langages de programmation, un objet occupe un ensemble d'emplacements contigus dans la mémoire de l'ordinateur. Un pointeur sur cet objet est tout simplement l'adresse du premier emplacement que l'objet occupe en mémoire ; pour accéder aux autres emplacements mémoire occupés par l'objet, il suffit d'incrémenter le pointeur.

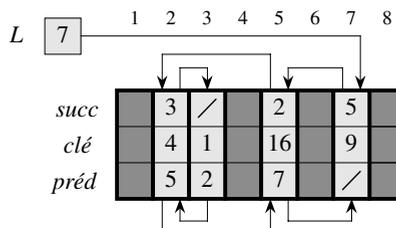


Figure 10.5 La liste chaînée de la figure 10.3(a) représentée par les tableaux *clé*, *succ* et *préd*. Chaque tranche verticale des tableaux représente un même objet. Les pointeurs stockés correspondent aux indices de tableau montrés au sommet ; les flèches montrent la façon dont on doit les interpréter. Les positions en gris clair contiennent les éléments de la liste. La variable L contient l'indice de la tête.

On peut employer la même stratégie pour l'implémentation des objets dans des environnements de programmation qui ne fournissent pas de types de données pointeur explicites. Par exemple, la figure 10.6 montre comment un tableau unique A peut servir à stocker la liste chaînée des figures 10.3(a) et 10.5. Un objet occupe un sous-tableau contigu $A[j..k]$. Chaque champ de l'objet correspond à un décalage dans l'intervalle 0 à $k - j$, et l'indice j est un pointeur sur l'objet. Dans la figure 10.6, les déplacements correspondant à *clé*, *succ* et *préd* valent respectivement 0, 1 et 2. Pour lire la valeur de *préd*[i], étant donné un pointeur i , on ajoute à la valeur i du pointeur le déplacement 2, ce qui nous permet de lire $A[i + 2]$.

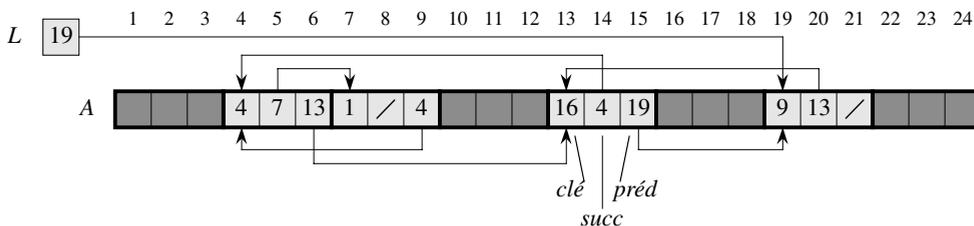


Figure 10.6 La liste chaînée des figures 10.3(a) et 10.5 représentée par un même tableau A . Chaque élément de la liste est un objet qui occupe un sous-tableau contigu de longueur 3 à l'intérieur du tableau. Les trois champs *clé*, *succ* et *préd* correspondent respectivement aux décalages 0, 1 et 2. L'indice du premier élément de l'objet est un pointeur sur cet objet. Les objets contenant les éléments de la liste sont en gris clair et les flèches montrent l'ordre de la liste.

La représentation par tableau unique est souple, au sens où elle permet à des objets de longueurs différentes d'être conservés dans le même tableau. La gestion d'une telle collection hétérogène d'objets est plus compliquée que celle d'une collection homogène, où tous les objets ont les mêmes champs. Comme la plupart des structures de données que nous considérerons sont composées d'éléments homogènes, la représentation par tableaux multiples sera bien suffisante pour nos besoins.

c) Allocation et libération des objets

Pour insérer une clé dans un ensemble dynamique représenté par une liste doublement chaînée, il faut allouer un pointeur sur un objet inutilisé dans la représentation de la liste chaînée. Il est donc utile de gérer, dans la représentation de la liste chaînée, le stockage des objets provisoirement inutilisés, de manière à pouvoir en allouer un en cas de besoin. Sur certains systèmes, un *récupérateur de place mémoire (garbage collector)* a pour rôle de déterminer les objets qui ne sont pas utilisés. Cependant, de nombreuses applications sont assez simples pour pouvoir endosser la responsabilité de retourner un objet non utilisé à un gestionnaire de stockage. Nous allons maintenant étudier le problème de l'allocation et de la libération d'objets homogènes en nous aidant de l'exemple d'une liste doublement chaînée représentée par des tableaux multiples.

Supposons que les tableaux servant à cette représentation aient pour longueur m et qu'à un certain moment l'ensemble dynamique contienne $n \leq m$ éléments. Il y a donc n objets qui représentent les éléments présents dans l'ensemble dynamique, les $m - n$ autres objets étant *libres* ; les objets libres peuvent servir à représenter des éléments qui seront insérés plus tard dans l'ensemble dynamique.

On conserve les objets libres dans une liste simplement chaînée, qu'on appelle *liste libre*. La liste libre n'utilise que le tableau *succ*, qui stocke les pointeurs *succ* de la liste. La tête de la liste libre est contenue dans la variable globale *libre*. Lorsque l'ensemble dynamique représenté par la liste chaînée L n'est pas vide, la liste libre peut être entrelacée avec la liste L , comme le montre la figure 10.7. On notera que chaque objet de la représentation est soit dans la liste L soit dans la liste libre, mais pas dans les deux à la fois.

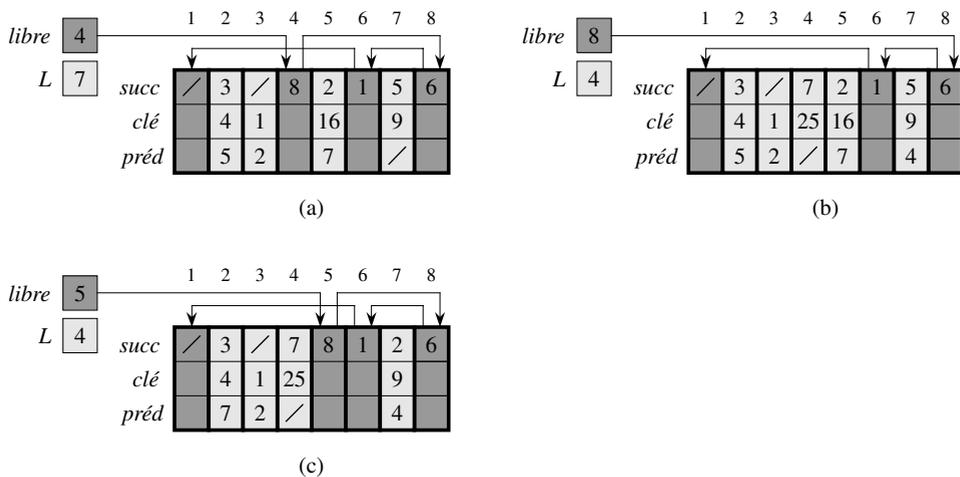


Figure 10.7 L'effet des procédures ALLOUER-OBJET et LIBÉRER-OBJET. (a) La liste de la figure 10.5 (en gris clair) et une liste libre (en gris foncé). Les flèches montrent la structure de la liste libre. (b) Le résultat de l'appel à ALLOUER-OBJET() (qui retourne l'indice 4), de l'initialisation de *clé*[4] à 25 et de l'appel à INSÉRER-LISTE(L , 4). La nouvelle tête de la liste libre est l'objet 8, qui était auparavant *succ*[4] dans la liste libre. (c) Après exécution de SUPPRIMER-LISTE(L , 5), on appelle LIBÉRER-OBJET(5). L'objet 5 devient la nouvelle tête de la liste libre, avec l'objet 8 pour successeur dans la liste libre.

La liste libre est une pile : le prochain objet qui sera alloué sera celui qui sera libéré en dernier. On peut utiliser une implémentation pour liste des opérations de pile EMPILER et DÉPILER pour mettre en œuvre respectivement les procédures d'allocation et de libération des objets. On suppose que la variable globale *libre* utilisée dans les procédures suivantes pointe sur le premier élément de la liste libre.

LIBÉRER-OBJET(x)

1 *succ*[x] \leftarrow *libre*

2 *libre* \leftarrow x

ALLOUER-OBJET()

```

1  si libre = NIL
2  alors erreur « plus assez d'espace disponible »
3  sinon x ← libre
4      libre ← succ[x]
5  retourner x
    
```

La liste libre contient initialement les n objets non alloués. Lorsque la liste libre est épuisée, la procédure ALLOUER-OBJET signale une erreur. Il est fréquent de mettre une même liste libre à la disposition de plusieurs listes chaînées. La figure 10.8 montre deux listes chaînées et une liste libre entrelacées, représentées par les tableaux clé, succ et préd.

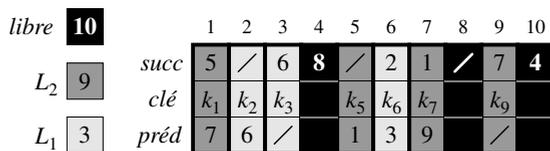


Figure 10.8 Deux listes chaînées, L₁ (en gris clair) et L₂ (en gris foncé) et une liste libre (en noir) entrelacées.

Les deux procédures s'exécutent en $O(1)$, ce qui les rend très pratiques. On peut les modifier pour qu'elles fonctionnent sur n'importe quelle collection d'objets homogènes, en faisant en sorte que l'un quelconque des champs de l'objet fasse office de champ succ de la liste libre.

Exercices

10.3.1 Dessiner une représentation de la séquence $\langle 13, 4, 8, 19, 5, 11 \rangle$ stockée sous la forme d'une liste doublement chaînée utilisant la représentation par tableaux multiples. Faire de même pour la représentation par tableau unique.

10.3.2 Écrire les procédures ALLOUER-OBJET et LIBÉRER-OBJET pour une collection homogène d'objets implémentés à l'aide de la représentation par tableau unique.

10.3.3 Pourquoi est-il inutile de (ré)initialiser le champ préd des objets dans l'implémentation des procédures ALLOUER-OBJET et LIBÉRER-OBJET ?

10.3.4 On souhaite souvent stocker tous les éléments d'une liste doublement chaînée de la manière la plus compacte possible ; on utilise, par exemple, les m premiers emplacements d'indice pour la représentation par tableaux multiples. (C'est le cas dans un environnement informatique à mémoire virtuelle paginée.) Expliquer comment implémenter les procédures

ALLOUER-OBJET et LIBÉRER-OBJET de manière à compacter la représentation. On supposera que, à l'extérieur de la liste elle-même, il n'existe aucun pointeur vers les éléments de la liste chaînée. (*Conseil* : utiliser l'implémentation par tableau d'une pile.)

10.3.5 Soit L une liste doublement chaînée de longueur m stockée dans des tableaux *clé*, *préd* et *succ* de longueur n . On suppose que ces tableaux sont gérés par les procédures ALLOUER-OBJET et LIBÉRER-OBJET qui gèrent une liste libre doublement chaînée F . On suppose aussi que, parmi les n éléments, m exactement sont dans la liste L et $n - m$ sont dans la liste libre. Écrire une procédure DENSIFIER-LISTE(L, F) qui, étant données la liste L et la liste libre F , déplace les éléments à l'intérieur de L pour qu'ils occupent les positions $1, 2, \dots, m$ du tableau et ajuste la liste libre F pour qu'elle occupe désormais les positions $m + 1, m + 2, \dots, n$. Le temps d'exécution de la procédure devra être en $\Theta(m)$, et la procédure ne devra utiliser qu'une quantité constante d'espace supplémentaire. Justifier soigneusement la validité de la procédure.

10.4 REPRÉSENTATION DES ARBORESCENCES

Les méthodes de représentation des listes données dans la section précédente s'étendent à n'importe quelle structure de données homogène. Dans cette section, nous nous intéressons particulièrement au problème de la représentation des arborescences à l'aide de structures de données chaînées. Nous commencerons par les arborescences binaires, puis nous présenterons une méthode pour les arborescences dont les nœuds peuvent avoir un nombre arbitraire d'enfants.

Chaque nœud est représenté par un objet. Comme pour les listes chaînées, on suppose que chaque nœud contient un champ *clé*. Les autres champs vitaux sont des pointeurs sur les autres nœuds et ils varient selon le type d'arborescence.

a) Arborescences binaires

Comme le montre la figure 10.9, on utilise les champs *p*, *gauche* et *droit* pour stocker les pointeurs vers le père, le fils gauche et le fils droit de chaque nœud d'un arbre binaire T . Si $p[x] = \text{NIL}$, alors x est la racine. Si le nœud x n'a pas de fils gauche, alors $\text{gauche}[x] = \text{NIL}$ et il en va de même pour le fils droit. La racine de l'arbre T est pointée par l'attribut $\text{racine}[T]$. Si $\text{racine}[T] = \text{NIL}$, alors l'arbre est vide.

b) Arborescences avec ramifications non bornées

Le schéma de représentation d'une arborescence binaire peut être étendu à n'importe quelle classe d'arbres pour laquelle le nombre de fils de chaque nœud vaut au plus une certaine constante k : on remplace les champs *gauche* et *droite* par $\text{fils}_1, \text{fils}_2, \dots, \text{fils}_k$. Ce schéma n'est plus valable quand le nombre de fils n'est pas borné, puisqu'on ne sait pas combien de champs (tableaux dans la représentation par tableaux multiples) allouer à l'avance. De plus, même si le nombre k de fils est borné

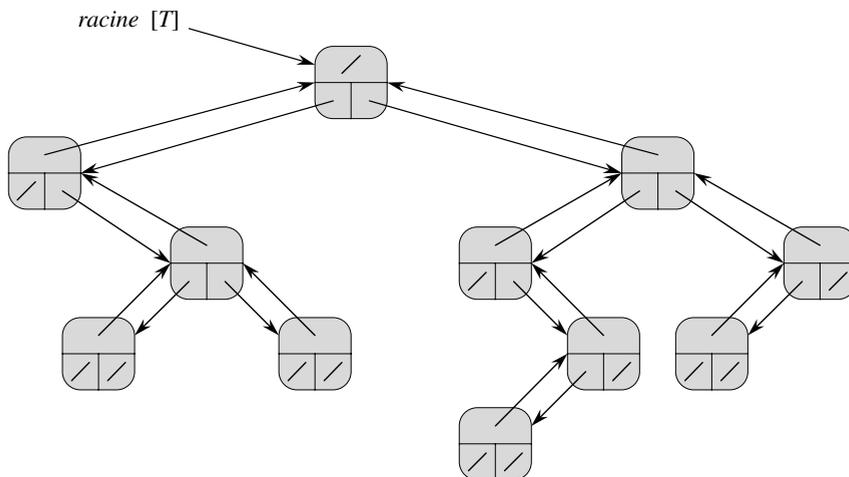


Figure 10.9 La représentation d'une arborescence binaire T . Chaque nœud x possède les champs $p[x]$ (parent), $gauche[x]$ (enfant de gauche), et $droite[x]$ (enfant de droite). Les champs *clé* n'apparaissent pas sur la figure.

par une grande constante, mais que la plupart des nœuds n'ont qu'un petit nombre de fils, on risque de gaspiller beaucoup de mémoire.

Heureusement, il existe un schéma astucieux permettant d'utiliser des arborescences binaires pour représenter des arborescences ayant un nombre arbitraire de fils. Il a l'avantage de n'utiliser qu'un espace en $O(n)$ pour une arborescence quelconque à n nœuds. La représentation *fils-gauche, frère-droit* est montrée à la figure 10.10. Comme précédemment, chaque nœud contient un pointeur p sur son père et $racine[T]$ pointe sur la racine de l'arborescence T . Toutefois, au lieu d'avoir un pointeur sur chacun de ses fils, chaque nœud x ne possède que deux pointeurs :

- 1) *fils-gauche* $[x]$ pointe sur le fils le plus à gauche du nœud x et
- 2) *frère-droite* $[x]$ pointe sur le frère de x situé immédiatement à sa droite.

Si le nœud x n'a aucun fils, alors *fils-gauche* $[x] = \text{NIL}$, et si x est le fils le plus à droite de son père, alors *frère-droit* $[x] = \text{NIL}$.

c) Autres représentations d'arborescences

Il arrive qu'on représente autrement les arborescences. Au chapitre 6, par exemple, nous avons représenté un tas, basé sur une arborescence binaire complète, par un tableau unique et un indice. Les arborescences que nous étudierons au chapitre 21 sont parcourues uniquement du bas vers le haut, ce qui permet de ne garder que les pointeurs parent ; il n'existe aucun pointeur vers un fils. De nombreux autres schémas sont possibles. Le meilleur choix dépend de l'application.

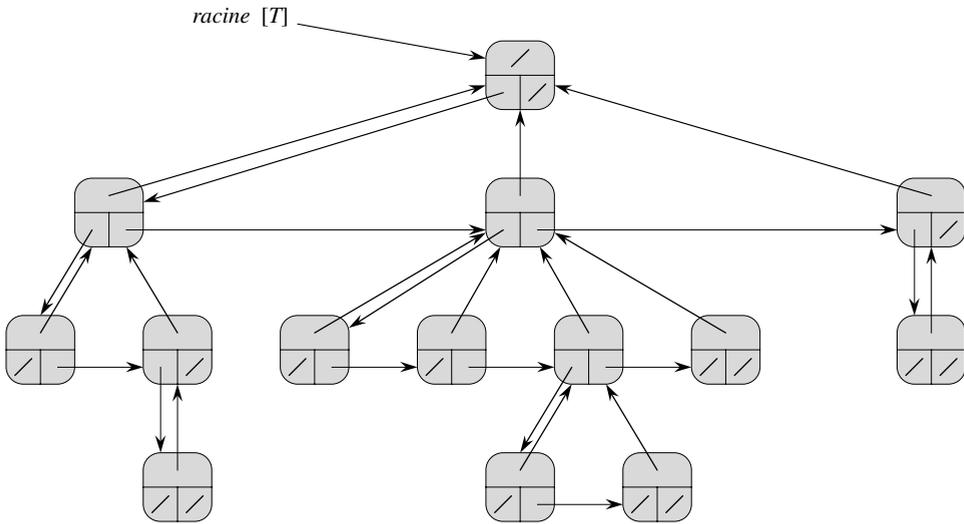


Figure 10.10 La représentation fils-gauche, frère-droite d'une arborescence T . Chaque nœud x possède les champs $p[x]$ (parent), $fils\text{-}gauche[x]$ (en bas à gauche) et $frère\text{-}droit[x]$ (en bas à droite). Les clés n'apparaissent pas sur la figure.

Exercices

10.4.1 Dessiner l'arborescence binaire dont l'indice de la racine est 6 et qui est représentée

indice	clé	gauche	droite
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

par les champs suivants.

10.4.2 Écrire une procédure récursive en $O(n)$ qui, étant donné une arborescence binaire à n nœuds, affiche la clé de chaque nœud de l'arborescence.

10.4.3 Écrire une procédure non-récursive en $O(n)$ qui, étant donné une arborescence binaire à n nœuds, affiche la clé de chaque nœud de l'arborescence. Utiliser une pile comme structure de données auxiliaire.

10.4.4 Écrire une procédure en $O(n)$ qui affiche toutes les clés d'une arborescence arbitraire à n nœuds, l'arborescence étant modélisée à l'aide de la représentation fils-gauche, frère-droit.

10.4.5 * Écrire une procédure non récursive en $O(n)$ qui, étant donné une arborescence binaire à n nœuds, affiche la clé de chaque nœud. On n'utilisera qu'un espace constant en plus de l'arborescence elle-même et l'on ne modifiera pas l'arborescence, même temporairement, durant la procédure.

10.4.6 * La représentation fils-gauche, frère-droit d'une arborescence arbitraire utilise trois pointeurs à chaque nœud : *fils-gauche*, *frère-droite* et *père*. Quel que soit le nœud, son parent est accessible et identifiable en temps constant ; tous ses enfants sont accessibles et identifiables en temps linéaire par rapport au nombre d'enfants. Montrer comment, en utilisant uniquement deux pointeurs et une valeur booléenne en chaque nœud, on peut atteindre et identifier le parent ou tous les enfants d'un nœud en un temps qui est linéaire par rapport au nombre d'enfants.

PROBLÈMES

10.1. Comparaisons entre listes

Pour chacun des quatre types de liste du tableau suivant, quel est le temps d'exécution asymptotique dans le cas le plus défavorable pour chacune des opérations d'ensembles dynamiques énumérées ?

	chaînée simple, non-triée	chaînée simple, triée	chaînée double, non-triée	chaînée double, triée
RECHERCHER(L, k)				
INSÉRER(L, x)				
SUPPRIMER(L, x)				
SUCCESEUR(L, x)				
PRÉDÉCESSEUR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10.2. Tas fusionnables avec des listes chaînées

Un *tas fusionnable* supporte les opérations suivantes : CONSTRUIRE-TAS (qui crée un tas vide), INSÉRER, MINIMUM, EXTRAIRE-MIN et UNION. ⁽¹⁾ Montrer comment implémenter les tas fusionnables à l'aide de listes chaînées dans chacun des cas suivants. Essayer de rendre chaque opération aussi efficace que possible. Analyser le

(1) Comme on a défini un tas fusionnable pour qu'il reconnaisse MINIMUM et EXTRAIRE-MIN, on peut parler de *tas min fusionnable*. Inversement, si le tas reconnaît MAXIMUM et EXTRAIRE-MAX, on parlera de *tas max fusionnable*.

temps d'exécution de chaque opération en fonction de la taille du ou des ensembles dynamiques sur lesquels elles agissent.

- a. Les listes sont triées.
- b. Les listes ne sont pas triées.
- c. Les listes ne sont pas triées et les ensembles dynamiques à fusionner sont disjoints.

10.3. Parcours d'une liste triée compacte

L'exercice 10.3.4 demandait de maintenir à jour une liste à n éléments confinés dans les n premières positions d'un tableau. On supposera que toutes les clés sont distinctes et que la liste compacte est également triée, autrement dit que $clé[i] < clé[succ[i]]$ pour tout $i = 1, 2, \dots, n$ tel que $succ[i] \neq \text{NIL}$. Partant de ces hypothèses, vous allez montrer que l'algorithme randomisé suivant peut servir à faire des recherches dans la liste avec un temps attendu $O(\sqrt{n})$.

RECHERCHE-LISTE-COMPACT(L, n, k)

```

1   $i \leftarrow tête[L]$ 
2  tant que  $i \neq \text{NIL}$  and  $clé[i] < k$ 
3      faire  $j \leftarrow \text{RANDOM}(1, n)$ 
4          si  $clé[i] < clé[j]$  and  $clé[j] \leq k$ 
5              alors  $i \leftarrow j$ 
6                  si  $clé[i] = k$ 
7                      alors retourner  $i$ 
8               $i \leftarrow succ[i]$ 
9  si  $i = \text{NIL}$  ou  $clé[i] > k$ 
10     alors retourner NIL
11    sinon retourner  $i$ 

```

Si l'on ignore les lignes 3–7 de la procédure, on a un algorithme classique de recherche dans une liste chaînée triée, l'indice i pointant vers chaque position de la liste à tour de rôle. La recherche se termine quand l'indice i « dépasse » la fin de la liste ou quand $clé[i] \geq k$. Dans ce dernier cas, si $clé[i] = k$, alors on a visiblement trouvé une clé ayant la valeur k . Si, en revanche, $clé[i] > k$, alors il n'existe aucune clé ayant la valeur k , auquel cas la seule chose à faire est de mettre immédiatement fin à la recherche.

Les lignes 3–7 essaient de sauter à une position j choisie aléatoirement. Un tel saut est intéressant si $clé[j]$ est supérieur à $clé[i]$ mais pas à k ; en pareil cas, j représente un emplacement de la liste que i aurait dû atteindre de la manière séquentielle classique. Comme la liste est compacte, on sait qu'un choix quelconque de j entre 1 et n indexe un objet de la liste, et non un emplacement de la liste libre.

Au lieu d'analyser les performances de RECHERCHE-LISTE-COMPACTE directement, on va analyser un algorithme voisin, RECHERCHE-LISTE-COMPACTE', qui

exécute deux boucles séparées. Cet algorithme prend un paramètre supplémentaire t qui définit un majorant pour le nombre d'itérations de la première boucle.

```

RECHERCHE-LISTE-COMPACTE'(L, n, k, t)
1  i ← tête[L]
2  pour q ← 1 à t
3    faire j ← RANDOM(1, n)
4        si clé[i] < clé[j] and clé[j] ≤ k
5            alors i ← j
6                si clé[i] = k
7                    alors retourner i
8  tant que i ≠ NIL et clé[i] < k
9    faire i ← succ[i]
10 si i = NIL ou clé[i] > k
11   alors retourner NIL
12   sinon retourner i

```

Pour comparer l'exécution des algorithmes RECHERCHE-LISTE-COMPACTE(L, k) et RECHERCHE-LISTE-COMPACTE'(L, k, t), on supposera que la suite d'entiers retournée par les appels RANDOM(1, n) est la même pour les deux algorithmes.

- a. Supposez que RECHERCHE-LISTE-COMPACTE(L, k) prenne t itérations de la boucle **tant que** des lignes 2–8. Prouver que RECHERCHE-LISTE-COMPACTE'(L, k, t) retourne la même réponse et que le nombre total d'itérations des deux boucles **pour** et **tant que** de RECHERCHE-LISTE-COMPACTE' est au moins t .

Dans l'appel RECHERCHE-LISTE-COMPACTE'(L, k, t), soit X_t la variable aléatoire qui décrit la distance, dans la liste chaînée (c'est-à-dire, à travers la chaîne des pointeurs *succ*), entre la position i et la clé désirée k , après t itérations de la boucle **pour** des lignes 2–7.

- b. Prouver que le temps d'exécution attendu de RECHERCHE-LISTE-COMPACTE'(L, k, t) est $O(t + E[X_t])$.
- c. Montrer que $E[X_t] \leq \sum_{r=1}^n (1 - r/n)^t$. (Conseil : Utiliser l'équation C.24).
- d. Montrer que $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$.
- e. Prouver que $E[X_t] \leq n/(t+1)$.
- f. Montrer que RECHERCHE-LISTE-COMPACTE'(L, k, t) s'exécute en un temps attendu de $O(t + n/t)$.
- g. En conclure que RECHERCHE-LISTE-COMPACTE s'exécute en un temps attendu de $O(\sqrt{n})$.
- h. Pourquoi suppose-t-on que toutes les clés sont distinctes dans RECHERCHE-LISTE-COMPACTE ? Prouver que les sauts aléatoires n'améliorent pas forcément le temps d'exécution asymptotique si la liste contient des doublons.

NOTES

Aho, Hopcroft et Ullman [6] et Knuth [182] sont d'excellentes références pour les structures de données élémentaires. Il existe beaucoup d'autres ouvrages qui traitent des structures de données fondamentales et de leurs implémentations dans tel ou tel langage de programmation. Citons, entre autres, Goodrich et Tamassia [128], Main [209], Shaffer [273] et Weiss [310, 312, 313]. Gonnet [126] fournit des données expérimentales sur les performances de nombreuses opérations de structure de données.

L'origine des piles et des files comme structures de données informatiques n'est pas certaine, puisque ces notions existaient déjà en mathématiques et dans les bureaux avant l'introduction des ordinateurs. Knuth [182] attribue à A. M. Turing le développement de piles pour gérer les liens entre sous-programmes en 1947.

Les structures de données basées sur les pointeurs semblent également provenir de la vie quotidienne. Selon Knuth, les pointeurs étaient apparemment utilisés dans les tout premiers ordinateurs avec mémoire à tambours. Le langage A-1 développé par G. M. Hopper en 1951 représentait les formules algébriques sous la forme d'arborescences binaires. Knuth attribue au langage IPL-II, développé en 1956 par A. Newell, J. C. Shaw et H. A. Simon, la le fait d'avoir reconnu l'importance des pointeurs et d'avoir popularisé leur emploi. Leur langage IPL-III, développé en 1957, incluait des opérations de pile explicites.

Chapitre 11

Tables de hachage

De nombreuses applications font appel à des ensembles dynamiques qui ne supportent que les opérations de dictionnaire INSÉRER, RECHERCHER et SUPPRIMER. Par exemple, un compilateur doit gérer une table de symboles, dans laquelle les clés des éléments sont des chaînes de caractères arbitraires qui correspondent aux identificateurs du langage. Une table de hachage est une structure de données permettant d'implémenter efficacement des dictionnaires. Bien que la recherche d'un élément dans une table de hachage puisse être aussi longue que la recherche d'un élément dans une liste chaînée ($\Theta(n)$ dans le cas le plus défavorable), en pratique le hachage est très efficace. Avec des hypothèses raisonnables, le temps moyen de recherche d'un élément dans une table de hachage est $O(1)$.

Une table de hachage est une généralisation de la notion simple de tableau ordinaire. L'adressage direct dans un tableau ordinaire utilise efficacement la possibilité d'examiner une position arbitraire dans un tableau en temps $O(1)$. La section 11.1 étudie plus en détail l'adressage direct. L'adressage direct est applicable lorsqu'on est en mesure d'allouer un tableau qui possède une position pour chaque clé possible.

Quand le nombre des clés effectivement stockées est petit comparé au nombre total des clés possibles, les tables de hachage remplacent efficacement l'adressage direct d'un tableau, puisqu'une table de hachage utilise généralement un tableau de taille proportionnelle au nombre des clés effectivement stockées. Au lieu d'utiliser la clé directement comme indice du tableau, l'indice est *calculé* à partir de la clé. La section 11.2 présentera les concepts fondamentaux, et notamment le « chaînage »

comme méthode de gestion des « collisions » (il y a collision quand plusieurs clés donnent le même indice de tableau). La section 11.3 expliquera comment des indices de tableau sont calculés à partir de clé via des fonctions de hachage. Nous présenterons et analyserons plusieurs variantes du mécanisme de base. La section 11.4 étudiera l'« adressage ouvert », qui est une autre technique de gestion des collisions. La conclusion de tout cela est que le hachage est une technique extrêmement efficace et commode : les opérations fondamentales de dictionnaire ne prennent qu'un temps $O(1)$ en moyenne. La section 11.5 expliquera comment le « hachage parfait » permet de faire de recherches avec un temps $O(1)$ dans le *cas le plus défavorable*, quand l'ensemble des clés stockées est statique (c'est-à-dire, quand il ne change plus une fois qu'il a été stocké).

11.1 TABLES À ADRESSAGE DIRECT

L'adressage direct est une technique simple qui fonctionne bien lorsque l'univers U des clés est raisonnablement petit. Supposons qu'une application ait besoin d'un ensemble dynamique dans lequel chaque élément possède une clé prise dans l'univers $U = \{0, 1, \dots, m - 1\}$, où m n'est pas trop grand. On supposera que deux éléments ne peuvent pas avoir la même clé.

Pour représenter l'ensemble dynamique, on utilise un tableau, aussi appelé *table à adressage direct*, $T[0..m - 1]$, dans lequel chaque position, ou *alvéole*, correspond à une clé de l'univers U . La figure 11.1 illustre cette approche ; l'alvéole k pointe vers un élément de l'ensemble ayant pour clé k . Si l'ensemble ne contient aucun élément de clé k , alors $T[k] = \text{NIL}$.

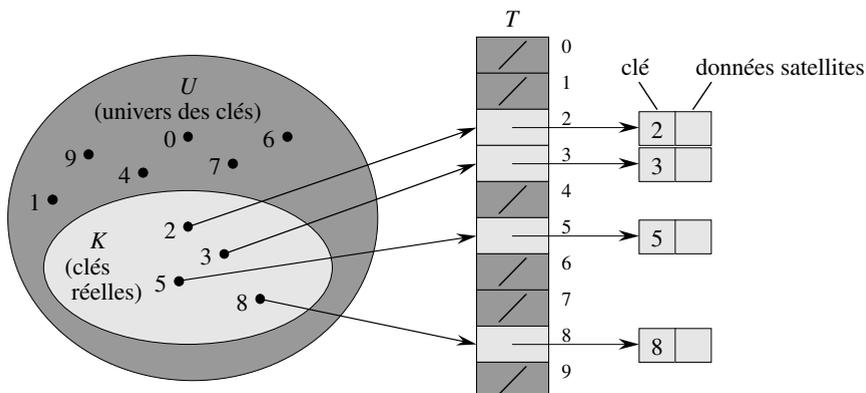


Figure 11.1 Implémentation d'un ensemble dynamique à l'aide d'une table à adressage direct T . Chaque clé de l'univers $U = \{0, 1, \dots, 9\}$ correspond à un indice de la table. L'ensemble $K = \{2, 3, 5, 8\}$ des clés réelles détermine les alvéoles de la table qui contiennent des pointeurs vers des éléments. Les autres alvéoles, en gris foncé, contiennent NIL.

L'implémentation des opérations de dictionnaire est triviale :

RECHERCHER-ADRESSAGE-DIRECT(T, k)

retourner $T[k]$

INSÉRER-ADRESSAGE-DIRECT(T, x)

$T[\text{clé}[x]] \leftarrow x$

SUPPRIMER-ADRESSAGE-DIRECT(T, x)

$T[\text{clé}[x]] \leftarrow \text{NIL}$

Chacune de ces opérations est rapide : un temps $O(1)$ suffit.

Pour certaines applications, les éléments de l'ensemble dynamique peuvent se trouver eux-mêmes dans la table à adressage direct. C'est-à-dire qu'au lieu de stocker la clé et les données satellite d'un objet en dehors de la table, avec un pointeur partant d'une alvéole de la table et pointant vers l'objet, on peut conserver l'objet entier dans l'alvéole, pour économiser de l'espace. Par ailleurs, il est souvent inutile de conserver le champ clé de l'objet, puisque si l'on dispose de l'indice d'un objet dans la table, on a aussi sa clé. Toutefois, si les clés ne sont pas conservées, il faut se ménager un moyen de savoir si l'alvéole est vide.

Exercices

11.1.1 Soit un ensemble dynamique S , représenté par une table à adressage direct T de longueur m . Décrire une procédure qui trouve l'élément maximal de S . Quelle est l'efficacité de votre procédure dans le cas le plus défavorable ?

11.1.2 Un *vecteur de bits* est tout simplement un tableau de bits (0 et 1). Un vecteur de bits de longueur m prend beaucoup moins d'espace qu'un tableau de m pointeurs. Décrire comment on pourrait utiliser un vecteur de bits pour représenter un ensemble dynamique d'éléments distincts sans données satellites. Les opérations de dictionnaire devront s'exécuter dans un temps $O(1)$.

11.1.3 Donner une suggestion d'implémentation d'une table à adressage direct dans laquelle les clés des éléments stockés ne sont pas nécessairement distinctes et où les éléments peuvent comporter des données satellites. Les trois opérations de dictionnaire (INSÉRER, SUPPRIMER et RECHERCHER) devront s'exécuter dans un temps $O(1)$. (Ne pas oublier que SUPPRIMER prend comme argument un pointeur vers un objet à supprimer, et non une clé.)

11.1.4 * On souhaite implémenter un dictionnaire en utilisant l'adressage direct sur un *très grand* tableau. Au départ, les entrées du tableau peuvent contenir des données quelconques ; l'initialisation complète du tableau s'avère peu pratique, à cause de sa taille. Décrire un schéma d'implémentation de dictionnaire via adressage direct sur un très grand tableau. Chaque objet stocké devra consommer un espace $O(1)$; les opérations RECHERCHER, INSÉRER et SUPPRIMER devront prendre chacune un temps $O(1)$; et l'initialisation des structures

de données devra se faire en un temps $O(1)$. (*Conseil* : Utiliser une pile supplémentaire, dont la taille est le nombre des clés effectivement stockées dans le dictionnaire, pour aider à déterminer si un élément donné du grand tableau est valide ou non.)

11.2 TABLES DE HACHAGE

L'inconvénient de l'adressage direct est évident : si l'univers U est grand, gérer une table T de taille $|U|$ peut se révéler compliqué, voire impossible, compte tenu de la mémoire généralement disponible dans un ordinateur. Par ailleurs, l'ensemble K des clés *réellement conservées* peut être tellement petit comparé à U que la majeure partie de l'espace alloué pour T est gaspillé.

Lorsque l'ensemble K des clés stockées dans un dictionnaire est beaucoup plus petit que l'univers U de toutes les clés possibles, une table de hachage requiert moins de place de stockage qu'une table à adressage direct. En particulier, les besoins en espace de stockage peuvent être réduits à $\Theta(|K|)$, bien que la recherche d'un élément dans la table de hachage s'effectue toujours en $O(1)$. (Le seul hic est que cette borne concerne le *temps moyen*, alors que pour l'adressage direct, elle est valable pour le *temps le plus défavorable*.)

Avec l'adressage direct, un élément de clé k est conservé dans l'alvéole k . Avec le hachage, cet élément est stocké dans l'alvéole $h(k)$; autrement dit, on utilise une **fonction de hachage** h pour calculer l'alvéole à partir de la clé k . Ici, h établit une correspondance entre l'univers U des clés et les alvéoles d'une **table de hachage** $T[0..m-1]$:

$$h : U \rightarrow \{0, 1, \dots, m-1\} .$$

On dit qu'un élément de clé k est **haché** dans l'alvéole $h(k)$; on dit également que $h(k)$ est la **valeur de hachage** de la clé k . La figure 11.2 en illustre le principe. Le but de la fonction de hachage est de réduire l'intervalle des indices de tableau à gérer. Au lieu de $|U|$ valeurs, il suffit de gérer m valeurs. Les besoins en stockage sont réduits en conséquence.

L'inconvénient de cette idée est que deux clés peuvent être hachées vers la même alvéole, entraînant ainsi une **collision**. Heureusement, il existe des techniques efficaces pour résoudre les conflits créés par les collisions.

Bien sûr, la solution idéale serait d'éviter complètement les collisions. On peut essayer d'atteindre ce but en choisissant de façon pertinente la fonction de hachage h . Une idée est de faire que h paraisse « aléatoire », ce qui permet d'éviter les collisions ou au moins d'en limiter le nombre. Le terme même de « hacher » traduit l'esprit de cette approche, en évoquant une découpe désordonnée en petits morceaux. (Bien sûr, une fonction de hachage h doit être déterministe au sens où une entrée donnée k doit toujours produire la même sortie $h(k)$.) Toutefois, comme $|U| > m$, il existe forcément au moins deux clés ayant la même valeur de hachage ; éviter complètement les collisions est donc impossible. Bien qu'une fonction de hachage

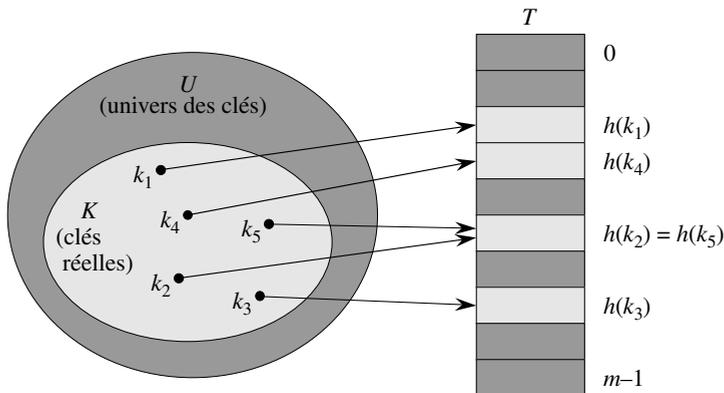


Figure 11.2 Utilisation d'une fonction de hachage h pour faire correspondre les clés à des alvéoles d'une table de hachage. Les clés k_2 et k_5 correspondent à la même alvéole et entrent donc en collision.

bien conçue, imitant « l'aléatoire », puisse minimiser le nombre de collisions, une méthode de résolution des collisions est donc indispensable.

La suite de cette section aborde la technique de résolution des collisions la plus simple, appelée chaînage. La section 11.4 introduit une autre méthode, appelée adressage ouvert.

a) Résolution des collisions par chaînage

Avec le **chaînage**, on place dans une liste chaînée tous les éléments hachés vers la même alvéole, comme le montre la figure 11.3. L'alvéole j contient un pointeur vers la tête de liste de tous les éléments hachés vers j ; si aucun n'élément n'est présent, l'alvéole j contient NIL.

Les opérations de dictionnaire sur une table de hachage T sont faciles à implémenter lorsque les collisions sont résolues par chaînage.

INSÉRER-HACHAGE-CHAÎNÉE(T, x)

insère x en tête de la liste $T[h(\text{clé}[x])]$

RECHERCHER-HACHAGE-CHAÎNÉE(T, k)

recherche un élément de clé k dans la liste $T[h(k)]$

SUPPRIMER-HACHAGE-CHAÎNÉE(T, x)

supprime x de la liste $T[h(\text{clé}[x])]$

Le temps d'exécution de l'insertion est $O(1)$ dans le cas le plus défavorable. La procédure d'insertion est rapide, en partie parce qu'elle suppose que l'élément x en cours d'insertion n'est pas déjà présent dans la table ; on peut, si nécessaire, vérifier cette hypothèse (moyennant un coût supplémentaire) en procédant à une recherche

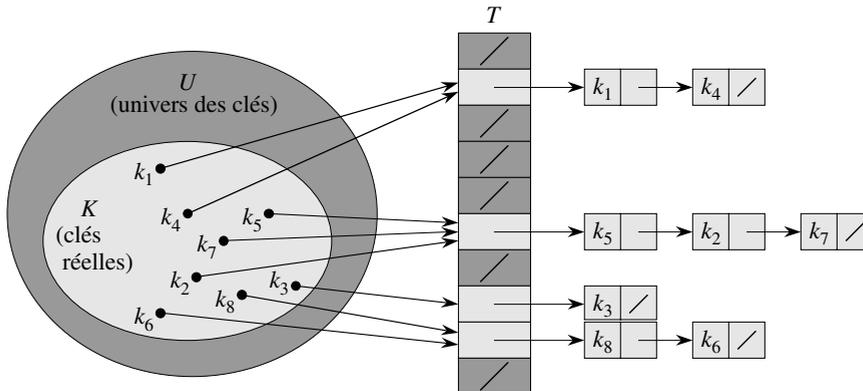


Figure 11.3 Résolution des collisions par chaînage. Chaque alvéole de la table de hachage $T[j]$ contient une liste chaînée de toutes les clés dont la valeur de hachage est j . Par exemple $h(k_1) = h(k_4)$ et $h(k_5) = h(k_2) = h(k_7)$.

avant insertion. Pour la recherche, le temps d'exécution du cas le plus défavorable est proportionnel à la longueur de la liste ; nous analyserons cette opération plus précisément un peu plus loin. La suppression d'un élément x peut se faire en temps $O(1)$ si les listes sont doublement chaînées. (Notez que SUPPRIMER-HACHAGE-CHAÎNÉE prend en entrée un élément x et non sa clé k , de sorte qu'il n'est pas nécessaire de commencer par chercher x . Si les listes étaient à chaînage simple, il ne serait pas d'une grande aide de prendre en entrée l'élément x au lieu de la clé k . Il faudrait encore trouver x dans la liste $T[h(\text{clé}[x])]$, de façon que le lien *succ* du prédécesseur de x puisse être réinitialisé comme il faut et contourner x . En pareil cas, suppression et recherche auraient fondamentalement le même temps d'exécution.)

b) Analyse du hachage avec chaînage

Quelle est l'efficacité du hachage avec chaînage ? En particulier, combien de temps faut-il pour rechercher un élément dont on connaît la clé ?

Étant donnée une table de hachage T à m alvéoles qui stocke n éléments, on définit pour T le **facteur de remplissage** α par n/m , c'est-à-dire le nombre moyen d'éléments stockés dans une chaîne. Notre analyse se fera en fonction de α qui peut être inférieur, égal ou supérieur à 1.)

Le comportement, dans le cas le plus défavorable, du hachage avec chaînage est très mauvais : les n clés sont toutes hachées vers la même alvéole, y formant une liste de longueur n . Le temps d'exécution de la recherche est alors $\Theta(n)$ plus le temps de calcul de la fonction de hachage ; pas mieux que si l'on avait utilisé une seule liste chaînée pour tous les éléments. Manifestement, les tables de hachage ne sont pas choisies pour leurs performances dans le cas le plus défavorable. (Le hachage parfait, présenté à la section 11.5, fournit toutefois de bonnes performances pour le cas le plus défavorable, quand l'ensemble des clés est statique.)

Les performances moyennes du hachage dépendent de la manière dont la fonction de hachage h répartit en moyenne l'ensemble des clés à stocker parmi les m alvéoles. La section 11.3 étudie les diverses possibilités, mais pour l'instant, on suppose que chaque élément a la même chance d'être haché vers l'une quelconque des alvéoles, indépendamment des endroits où les autres éléments sont allés. Cette hypothèse est dite de *hachage uniforme simple*.

Pour $j = 0, 1, \dots, m - 1$, notons la longueur de la liste $T[j]$ par n_j , de sorte que

$$n = n_0 + n_1 + \dots + n_{m-1}, \quad (11.1)$$

et la valeur moyenne de n_j est $E[n_j] = \alpha = n/m$.

On suppose que la valeur de hachage $h(k)$ peut être calculée en temps $O(1)$, de sorte que le temps requis par la recherche d'un élément de clé k dépend linéairement de la longueur de la liste $T[h(k)]$. En plus du temps $O(1)$ requis par le calcul de la fonction de hachage et l'accès à l'alvéole $h(k)$, on considère le nombre attendu d'éléments examinés par l'algorithme de recherche, c'est-à-dire le nombre d'éléments de la liste $T[h(k)]$ qui sont testés pour voir si leur clé est égale à k . On considèrera deux cas. Dans le premier, la recherche échoue : aucun élément de la table ne possède la clé k . Dans le second, la recherche permet de trouver un élément de clé k .

Théorème 11.1 *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une recherche infructueuse prend un temps moyen $\Theta(1 + \alpha)$, sous l'hypothèse d'un hachage uniforme simple.*

Démonstration : Si l'on suppose que le hachage est uniforme simple, toute clé k a des chances égales d'être hachée vers l'une quelconque des m alvéoles. Le temps moyen pour la recherche infructueuse d'une clé k est le temps moyen pour la poursuite de la recherche jusqu'à la fin de la liste $T[h(k)]$, qui a une longueur moyenne $E[n_{h(k)}] = \alpha$. Donc, le nombre moyen d'éléments examinés dans une recherche infructueuse est α , et le temps total requis (y compris le temps de calcul de $h(k)$) est $\Theta(1 + \alpha)$. \square

La situation pour une recherche réussie est quelque peu différente, vu que chaque liste n'a pas la même probabilité d'être examinée. Ici, la probabilité qu'une liste soit examinée est proportionnelle au nombre de ses éléments. Néanmoins, le temps de recherche moyen est encore $\Theta(1 + \alpha)$.

Théorème 11.2 *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une recherche réussie prend en moyenne un temps $\Theta(1 + \alpha)$, sous l'hypothèse d'un hachage uniforme simple.*

Démonstration : On suppose que l'élément recherché a une probabilité égale d'être l'un quelconque des n éléments stockés dans la table. Le nombre d'éléments examinés lors d'une recherche fructueuse portant sur un élément x est égal à 1 plus le nombre d'éléments qui apparaissent avant x dans la liste de x . Les éléments placés avant x dans la liste ont tous été insérés après x , vu que les nouveaux éléments sont placés au début

de la liste. Pour trouver le nombre moyen d'éléments examinés, on prend la moyenne sur les n éléments x de la table, de 1 plus le nombre moyen d'éléments ajoutés à la liste de x après que x a été ajouté à la liste. Soit x_i le i ème élément inséré dans la table, avec $i = 1, 2, \dots, n$, et soit $k_i = \text{clé}[x_i]$. Pour les clés k_i et k_j , on définit la variable indicatrice $X_{ij} = \mathbb{1}\{h(k_i) = h(k_j)\}$. En supposant qu'il y a hachage uniforme simple, on a $\Pr\{h(k_i) = h(k_j)\} = 1/m$, et donc d'après le lemme 5.1, $E[X_{ij}] = 1/m$. Donc, le nombre moyen d'éléments examinés dans une recherche réussie est

$$\begin{aligned}
 E \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij} \right) \right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}] \right) \quad (\text{d'après la linéarité de l'espérance}) \\
 &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m} \right) \\
 &= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i) \\
 &= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i \right) \\
 &= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right) \quad (\text{d'après l'équation (A.1)}) \\
 &= 1 + \frac{n-1}{2m} \\
 &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
 \end{aligned}$$

Ainsi, le temps total requis par une recherche réussie (y compris le temps pour calculer la fonction de hachage) est $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. \square

Que signifie cette analyse ? Si le nombre d'alvéoles de la table de hachage est au moins proportionnel au nombre d'éléments de la table, on a $n = O(m)$ et, par conséquent, $\alpha = n/m = O(m)/m = O(1)$. Donc, la recherche prend un temps constant en moyenne. Comme l'insertion prend un temps $O(1)$ dans le cas le plus défavorable, et que la suppression prend un temps $O(1)$ dans le cas le plus défavorable si les listes sont doublement chaînées, toutes les opérations de dictionnaire peuvent donc se faire en temps $O(1)$ en moyenne.

Exercices

11.2.1 Supposons qu'on utilise une fonction de hachage h pour hacher n clés distinctes vers un tableau T de longueur m . Si le hachage est uniforme simple, quel est le nombre moyen de collisions ? Plus précisément, quelle est la cardinalité attendue de $\{(k, l) : k \neq l \text{ et } h(k) = h(l)\}$?

11.2.2 Montrer comment on réalise l'insertion des clés 5, 28, 19, 15, 20, 33, 12, 17, 10 dans une table de hachage où les collisions sont résolues par chaînage. On suppose que la table contient 9 alvéoles et que la fonction de hachage est $h(k) = k \bmod 9$.

11.2.3 Le professeur Moriarty pense qu'on peut obtenir des gains de performance substantiels si l'on modifie le schéma de chaînage de telle manière que chaque liste soit triée. Comment la modification du professeur affecte-t-elle le temps d'exécution des recherches réussies, des recherches infructueuses, des insertions et des suppressions ?

11.2.4 Proposer une façon d'allouer et libérer l'espace de stockage des éléments à l'intérieur de la table elle-même en chaînant toutes les alvéoles inutilisées pour former une liste libre. On supposera qu'une alvéole peut contenir un indicateur plus, soit un élément et un pointeur, soit deux pointeurs. Toutes les opérations de dictionnaire et de liste libre devront s'exécuter en temps $O(1)$. La liste libre doit-elle être doublement chaînée, ou une chaîne simple suffit-elle ?

11.2.5 Montrer que, si $|U| > nm$, il existe un sous-ensemble de U de taille n qui est constitué de clés toutes hachées vers la même alvéole ; de sorte que le temps de recherche, dans le cas le plus défavorable, pour le hachage avec chaînage est $\Theta(n)$.

11.3 FONCTIONS DE HACHAGE

Dans cette section, nous étudierons certaines possibilités quant à la conception de bonnes fonctions de hachage, puis nous présenterons trois schémas de création. Deux de ces mécanismes, le hachage par division et le hachage par multiplication, sont heuristiques par nature, alors que le troisième mécanisme, le hachage universel, utilise la randomisation pour offrir des performances bonnes et prouvables comme telles.

a) *Qu'est-ce qui fait une bonne fonction de hachage ?*

Une bonne fonction de hachage vérifie (approximativement) l'hypothèse du hachage uniforme simple : chaque clé a autant de chances d'être hachée vers l'une quelconque des m alvéoles, indépendamment des endroits où sont allées les autres clés. Malheureusement, il est impossible en général de vérifier cette condition ; en effet, il est rare que l'on connaisse la distribution de probabilité selon laquelle les clés sont tirées, et les clés peuvent ne pas être tirées de façon indépendante.

Il peut advenir, à l'occasion, que l'on connaisse la distribution. Par exemple, supposons que les clés soient k nombres réels aléatoires, distribués indépendamment et uniformément dans l'intervalle $0 \leq k < 1$. Dans ce cas, la fonction de hachage

$$h(k) = \lfloor km \rfloor$$

satisfait à la condition du hachage uniforme simple.

En pratique, on peut souvent faire appel à des techniques heuristiques pour créer une fonction de hachage efficace. On a parfois besoin, pour ce processus de conception, de connaître des informations qualitatives sur la distribution des clés. Considérons, par exemple, la table des symboles d'un compilateur, dans laquelle les clés sont des chaînes de caractères arbitraires représentant les identificateurs d'un programme. Il arrive souvent que des symboles proches, comme `pt` et `pts`, soient employés dans le même programme. Une bonne fonction de hachage minimise le risque que ces variantes se retrouvent dans la même alvéole.

Une approche courante est de former la valeur de hachage de manière à la rendre indépendante des motifs (*patterns*) susceptibles d'exister dans les données. Par exemple, la « méthode de la division » (étudiée plus loin) calcule la valeur de hachage comme reste de la division de la clé par un nombre premier particulier. A moins qu'il n'existe des relations entre ce nombre premier et des motifs figurant dans la distribution des clés, cette méthode donne de bons résultats.

Finalement, on remarque que certaines applications des fonctions de hachage risquent d'exiger des propriétés plus fortes que le hachage uniforme simple. Par exemple, on pourrait souhaiter que les clés qui sont « proches » dans un certain sens génèrent des valeurs de hachage très éloignées les unes des autres. (Cette propriété est particulièrement souhaitable quand on utilise le sondage linéaire, défini à la section 11.4.) Le hachage universel, présenté à la section 11.3.3, offre souvent les propriétés souhaitées.

b) Clés interprétées comme des entiers naturels

La plupart des fonctions de hachage supposent que l'univers des clés est l'ensemble $\mathbf{N} = \{0, 1, 2, \dots\}$ des entiers naturels. Donc, si les clés ne sont pas des entiers naturels, on doit trouver un moyen de les interpréter comme des entiers naturels. Par exemple, une clé sous forme de chaîne de caractères peut être interprétée comme un entier exprimé dans une base adaptée. Ainsi, l'identificateur `pt` pourrait être interprété comme la paire d'entiers décimaux (112, 116), puisque `p` = 112 et `t` = 116 dans l'ensemble des caractères ASCII ; ensuite, en l'exprimant en base 128, `pt` devient $(112 \cdot 128) + 116 = 14452$. Le plus souvent, il est facile de trouver une méthode aussi simple pour une application donnée quelconque, permettant d'interpréter chaque clé comme un entier naturel (potentiellement grand). À partir de maintenant, nous supposerons que les clés sont des entiers naturels.

11.3.1 Méthode de la division

Dans la *méthode de la division*, pour créer des fonctions de hachage, on fait correspondre une clé k avec l'une des m alvéoles en prenant le reste de la division de k par m . En d'autres termes, la fonction de hachage est

$$h(k) = k \bmod m .$$

Par exemple, si la table de hachage a une taille $m = 12$ et que la clé est $k = 100$, alors $h(k) = 4$. Comme il ne demande qu'une seule opération de division, le hachage par division est très rapide.

Lorsqu'on utilise la méthode de la division, on évite en général certaines valeurs de m . Par exemple, m ne doit pas être une puissance de 2 car, si $m = 2^p$, $h(k)$ est constitué des p bits de poids faible de k . À moins que l'on ne sache que tous les motifs des p bits d'ordre inférieur sont équiprobables, il vaut mieux faire dépendre la fonction de hachage de tous les bits de la clé. Comme l'exercice 11.3.3 vous demandera de le montrer, choisir $m = 2^p - 1$ quand k est une chaîne de caractères interprétée en base 2^p , risque d'être un choix malheureux, car la permutation des caractères de k ne modifie pas sa valeur de hachage. Un nombre premier pas trop proche d'une puissance exacte de 2 est souvent un bon choix pour m . Par exemple, supposons qu'on souhaite allouer une table de hachage, avec résolution des collisions par chaînage, pour gérer environ $n = 2000$ chaînes de caractères, avec des caractères sur 8 bits. L'examen de 3 éléments en moyenne en cas de recherche infructueuse n'est pas catastrophique et on alloue donc une table de hachage de taille $m = 701$. On prend le nombre 701 parce qu'il est premier et proche de $2000/3$, sans être proche d'une puissance de 2. Si l'on traite chaque clé k comme un entier, notre fonction de hachage devient

$$h(k) = k \bmod 701 .$$

11.3.2 Méthode de la multiplication

La *méthode de la multiplication* pour créer des fonctions de hachage agit en deux étapes. D'abord, on multiplie la clé k par une constante A de l'intervalle $0 < A < 1$ et on extrait la partie décimale de kA . Ensuite, on multiplie cette valeur par m et on prend la partie entière du résultat. En résumé, la fonction de hachage vaut

$$h(k) = \lfloor m(kA \bmod 1) \rfloor ,$$

où « $kA \bmod 1$ » représente la partie décimale de kA , c'est-à-dire $kA - \lfloor kA \rfloor$.

Un avantage de la méthode par multiplication est que la valeur de m n'est pas critique. On choisit en général une puissance de 2 ($m = 2^p$ pour un certain entier p) ce qui permet d'implémenter facilement la fonction sur la plupart des ordinateurs de la manière suivante : supposons que la taille du mot sur la machine concernée soit w bits et que k tienne dans un seul mot. On limite A à une fraction de la forme $s/2^w$, où s est un entier de la plage $0 < s < 2^w$. En se référant à la figure 11.4, on commence par multiplier k par l'entier à w bits $s = A \cdot 2^w$. Le résultat est une valeur à $2w$ bits $r_1 2^w + r_0$, où r_1 est le mot de poids fort du produit et r_0 est le mot de poids faible du produit. La valeur de hachage à p bits désirée se compose des p bits les plus significatifs de r_0 .

Bien que cette méthode fonctionne pour n'importe quelle valeur de la constante A , elle fonctionne mieux pour certaines valeurs que pour d'autres. Le meilleur choix

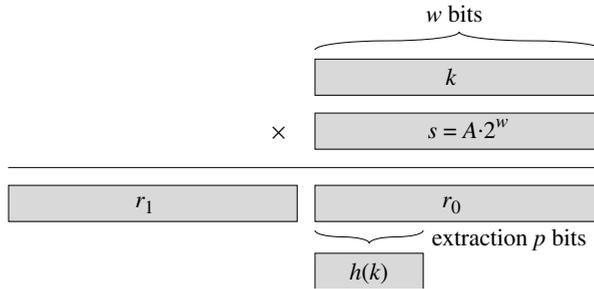


Figure 11.4 La méthode de hachage par multiplication. La représentation w bits de la clé k est multipliée par la valeur w bits $s = A \cdot 2^w$. Les p bits de poids faible de la moitié inférieure (w bits) du produit forment la valeur de hachage désirée $h(k)$.

dépend des caractéristiques des données à hacher. Knuth [185] suggère que

$$A \approx (\sqrt{5} - 1)/2 = 0,6180339887 \dots \quad (11.2)$$

a de bonnes chances de bien marcher.

Par exemple, prenons $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ et $w = 32$. En reprenant l'idée de Knuth, on choisit pour A la fraction de la forme $s/2^{32}$ qui est la plus proche de $(\sqrt{5} - 1)/2$, de sorte que $A = 2654435769/2^{32}$. Alors, $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, et donc $r_1 = 76300$ et $r_0 = 17612864$. Les 14 bits les plus significatifs de r_0 donnent la valeur $h(k) = 67$.

11.3.3 Hachage universel

Si un ennemi choisit les clés à hacher via une fonction de hachage figée, il peut choisir n clés qui sont toutes hachées vers la même alvéole, ce qui entraîne un temps de recherche de $\Theta(n)$ en moyenne. Toute fonction de hachage fixée à l'avance est vulnérable à ce type de comportement de cas le plus défavorable ; le seul moyen efficace d'améliorer cette situation est de choisir la fonction de hachage *aléatoirement*, c'est-à-dire indépendamment des clés qui vont être réellement stockées. Cette approche, appelée *hachage universel*, donne de bonnes performances en moyenne, quelles que soient les clés choisies par l'ennemi.

Le principe du hachage universel est de sélectionner la fonction de hachage au hasard, à partir d'une classe de fonctions soigneusement conçue, au début de l'exécution. Comme pour le tri rapide, la randomisation garantit qu'aucune entrée particulière ne provoquera systématiquement le comportement du cas le plus défavorable. A cause de la randomisation, l'algorithme peut se comporter différemment à chaque exécution, même sur une entrée identique ; cette approche garantit de bonnes performances en moyenne, quelles que soient les clés fournies en entrée. Si l'on revient à l'exemple de la table des symboles d'un compilateur, on s'aperçoit que le choix du programmeur pour les identificateurs ne peut plus maintenant provoquer systématiquement de mauvaises performances de hachage. Ces mauvaises performances

n'apparaissent que si le compilateur choisit une fonction de hachage aléatoire qui provoque le mauvais hachage de l'ensemble des identificateurs, mais la probabilité pour que cela se produise est faible et est la même pour n'importe quel ensemble d'identificateurs de la même taille.

Soit \mathcal{H} une collection finie de fonctions de hachage qui créent une correspondance entre un univers U de clés et l'intervalle $\{0, 1, \dots, m - 1\}$. On dit d'une telle collection qu'elle est **universelle** si, pour chaque paire de clés $k, l \in U$ distinctes, le nombre de fonctions de hachage $h \in \mathcal{H}$ pour lesquelles $h(k) = h(l)$ vaut au plus $|\mathcal{H}|/m$. Autrement dit, avec une fonction de hachage choisie aléatoirement dans \mathcal{H} , les chances de collision entre des clés distinctes k et l ne sont pas plus grandes que la chance $1/m$ d'avoir une collision si $h(k)$ et $h(l)$ sont choisies aléatoirement et indépendamment dans l'ensemble $\{0, 1, \dots, m - 1\}$.

Le théorème suivant montre qu'une classe universelle de fonctions de hachage offre un bon comportement dans le cas moyen. Rappelons-nous que n_i désigne la longueur de la liste $T[i]$.

Théorème 11.3 *Supposons qu'une fonction de hachage h soit choisie dans une collection universelle de fonctions de hachages et utilisée pour hacher n clés vers une table T de taille m , avec emploi du chaînage pour gérer les collisions. Si la clé k n'est pas dans la table, alors la longueur moyenne $E[n_{h(k)}]$ de la liste vers laquelle la clé k est hachée est d'au plus α . Si la clé k est dans la table, alors la longueur moyenne $E[n_{h(k)}]$ de la liste contenant la clé k est d'au plus $1 + \alpha$.*

Démonstration : Notez que les espérances ici sont calculées à partir du choix de la fonction de hachage, et qu'elles ne dépendent d'aucunes hypothèses concernant la distribution des clés. Pour toute paire k et l de clés distinctes, on définit la variable indicatrice $X_{kl} = I\{h(k) = h(l)\}$. Comme, d'après la définition, une paire de clés a une probabilité de collision d'au plus $1/m$, on a $\Pr\{h(k) = h(l)\} \leq 1/m$, et donc le lemme 5.1 entraîne que $E[X_{kl}] \leq 1/m$.

Définissons ensuite, pour chaque clé k , la variable aléatoire Y_k qui est égale au nombre de clés autres que k qui sont hachées vers l'alvéole de k , de sorte que

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

On a donc

$$\begin{aligned} E[Y_k] &= E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \quad (\text{d'après la linéarité de l'espérance}) \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}. \end{aligned}$$

Le reste de la démonstration dépend de savoir si la clé k est dans la table T .

- Si $k \notin T$, alors $n_{h(k)} = Y_k$ et $|\{l : l \in T \text{ et } l \neq k\}| = n$. Donc, $E[n_{h(k)}] = E[Y_k] \leq n/m = \alpha$.
- Si $k \in T$, alors comme la clé k figure dans la liste $T[h(k)]$ et que le compteur Y_k n'inclut pas la clé k , on a $n_{h(k)} = Y_k + 1$ et $|\{l : l \in T \text{ et } l \neq k\}| = n - 1$. Donc $E[n_{h(k)}] = E[Y_k] + 1 \leq (n - 1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$. \square

Le corollaire suivant dit que le hachage universel fournit le résultat désiré : il est désormais impossible à un ennemi de sélectionner une suite d'opérations qui force le temps d'exécution le plus défavorable. En randomisant intelligemment le choix de la fonction de hachage au moment de l'exécution, on a la garantie que toute séquence d'opérations pourra être traitée avec un bon temps attendu d'exécution.

Corollaire 11.4 *En combinant hachage universel et gestion des collisions par chaînage dans une table à m alvéoles, il faut un temps moyen $\Theta(n)$ pour traiter une suite quelconque de n opérations INSÉRER, RECHERCHER et SUPPRIMER contenant $O(m)$ opérations INSÉRER.*

Démonstration : Comme le nombre d'insertions est $O(m)$, on a $n = O(m)$ et donc $\alpha = O(1)$. Les opérations INSÉRER et SUPPRIMER prennent un temps constant et, d'après le théorème 11.3, le temps moyen de chaque opération RECHERCHER est $O(1)$. D'après la linéarité de l'espérance, le temps moyen pour l'ensemble de la séquence d'opérations est donc $O(n)$. \square

a) Conception d'une classe universelle de fonctions de hachage

Il est assez simple de concevoir une classe universelle de fonctions de hachages, comme nous le verrons en nous aidant d'un peu de théorie des nombres. Commencez par relire le chapitre 31 si vous n'êtes pas très familier de la théorie des nombres.

On commence par choisir un nombre premier p assez grand pour que toute clé possible k soit dans l'intervalle 0 à $p - 1$, bornes comprises. Soit \mathbf{Z}_p l'ensemble $\{0, 1, \dots, p - 1\}$, et soit \mathbf{Z}_p^* l'ensemble $\{1, 2, \dots, p - 1\}$. Comme p est premier, on peut résoudre les équations modulo p avec les méthodes vues au chapitre 31. Comme on suppose que la taille de l'univers des clés est supérieure au nombre d'alvéoles de la table de hachage, on a $p > m$.

Définissons maintenant la fonction de hachage $h_{a,b}$ pour tout $a \in \mathbf{Z}_p^*$ et tout $b \in \mathbf{Z}_p$, en utilisant une transformation linéaire suivie de réductions modulo p puis modulo m :

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m. \quad (11.3)$$

Par exemple, avec $p = 17$ et $m = 6$, on a $h_{3,4}(8) = 5$. La famille de toutes ces fonctions de hachages est

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^* \text{ et } b \in \mathbf{Z}_p\}. \quad (11.4)$$

Chaque fonction de hachage $h_{a,b}$ définit une correspondance de \mathbf{Z}_p vers \mathbf{Z}_m . Cette classe de fonctions de hachage présente l'intéressante propriété que la taille m de l'intervalle de sortie est arbitraire (et pas forcément une valeur première) ; nous ferons

usage de cette caractéristique à la section 11.5. Comme il y a $p - 1$ choix pour a et p choix pour b , il y a $p(p - 1)$ fonctions de hachages de $\mathcal{H}_{p,m}$.

Théorème 11.5 *La classe $\mathcal{H}_{p,m}$ de fonctions de hachages définie par les équations (11.3) et (11.4) est universelle.*

Démonstration : Soient deux clés distinctes k et l de \mathbf{Z}_p , de sorte que $k \neq l$. Pour une fonction de hachage donnée $h_{a,b}$, soit

$$r = (ak + b) \bmod p,$$

$$s = (al + b) \bmod p.$$

Notez, pour commencer, que $r \neq s$. Pourquoi ? Observez que

$$r - s \equiv a(k - l) \pmod{p}.$$

Il s'ensuit que $r \neq s$ parce que p est premier et parce que a et $(k - l)$ sont tous les deux non nuls modulo p , ce qui fait que leur produit est forcément non nul modulo p d'après le théorème 31.6. Par conséquent, lors du calcul d'un quelconque $h_{a,b}$ de $\mathcal{H}_{p,m}$, à des entrées distinctes k et l sont associées des valeurs distinctes r et s modulo p ; il n'y a pas de collisions, tout au moins au « niveau modulo p ». En outre, chacun des $p(p - 1)$ choix possibles pour la paire (a, b) avec $a \neq 0$ donne une paire résultante (r, s) différente avec $r \neq s$, car on peut résoudre a et b à partir de r et s :

$$a = ((r - s)((k - l)^{-1} \bmod p)) \bmod p,$$

$$b = (r - ak) \bmod p,$$

où $((k - l)^{-1} \bmod p)$ désigne l'inverse multiplicatif unique, modulo p , de $k - l$. Comme il n'y a que $p(p - 1)$ paires possibles (r, s) avec $r \neq s$, il existe une bijection entre paires (a, b) avec $a \neq 0$ et paires (r, s) avec $r \neq s$. Donc, pour toute paire donnée d'entrées k et l , si l'on choisit (a, b) aléatoirement de manière uniforme dans $\mathbf{Z}_p^* \times \mathbf{Z}_p$, alors la paire résultante (r, s) a la même probabilité d'être l'une quelconque des paires de valeurs distinctes modulo p .

Il s'ensuit que la probabilité que des clés distinctes k et l entrent en collision est égale à la probabilité que $r \equiv s \pmod{m}$ quand r et s sont choisis aléatoirement comme valeurs distinctes modulo p . Pour une valeur donnée de r , parmi les $p - 1$ autres valeurs possible de s , le nombre de valeurs s telles que $s \neq r$ et $s \equiv r \pmod{m}$ est au plus

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p + m - 1)/m) - 1 \quad (\text{d'après l'inégalité (3.6)}) \\ &= (p - 1)/m. \end{aligned}$$

La probabilité que s entre en collision avec r , après réduction modulo m , est au plus égale à $((p - 1)/m)/(p - 1) = 1/m$.

En conclusion, pour toute paire de valeurs distinctes $k, l \in \mathbf{Z}_p$,

$$\Pr \{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m,$$

et donc $\mathcal{H}_{p,m}$ est bien universelle. □

Exercices

11.3.1 Supposons qu'on souhaite parcourir une liste chaînée de longueur n , dans laquelle chaque élément contient une clé k en plus d'une valeur de hachage $h(k)$. Chaque clé est une chaîne de caractères longue. Comment pourrait-on tirer parti des valeurs de hachage pendant la recherche d'un élément de clé donnée ?

11.3.2 Supposons qu'une chaîne de r caractères soit hachée dans m alvéoles en étant traitée comme un nombre en base 128 auquel on applique la méthode de la division. Le nombre m est facilement représenté par un mot mémoire sur 32 bits, mais la chaîne de r caractères, traitée comme un nombre en base 128, occupe plusieurs mots. Comment peut-on appliquer la méthode de la division pour calculer la valeur de hachage de la chaîne de caractères sans utiliser plus qu'un nombre de mots constant pour le stockage, en dehors de la chaîne elle-même ?

11.3.3 On considère une variante de la méthode de la division dans laquelle $h(k) = k \bmod m$, où $m = 2^p - 1$ et k est une chaîne de caractères interprétée en base 2^p . Montrer que si la chaîne x peut être déduite de la chaîne y par permutation de ses caractères, alors x et y ont même valeur de hachage. Donner un exemple d'application pour laquelle cette propriété de la fonction de hachage serait indésirable.

11.3.4 On considère une table de hachage de taille $m = 1000$ et la fonction de hachage $h(k) = \lfloor m(kA \bmod 1) \rfloor$ pour $A = (\sqrt{5} - 1)/2$. Calculer les emplacements correspondant aux clés 61, 62, 63, 64 et 65.

11.3.5 * On dit qu'une famille \mathcal{H} de fonctions de hachage reliant un ensemble fini U à un ensemble fini B est ε -universelle si, pour toute paire d'éléments distincts k et l de U ,

$$\Pr \{h(k) = h(l)\} \leq \varepsilon,$$

la probabilité étant défini par le tirage aléatoire de la fonction de hachage h dans la famille \mathcal{H} . Montrer qu'une famille ε -universelle de fonctions de hachage doit vérifier

$$\varepsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

11.3.6 * Soit U l'ensemble des n -uplets dont les valeurs sont tirées de \mathbf{Z}_p , et soit $B = \mathbf{Z}_p$, avec p premier. Définir la fonction de hachage $h_b : U \rightarrow B$ pour $b \in \mathbf{Z}_p$ sur un n -uplet en entrée $\langle a_0, a_1, \dots, a_{n-1} \rangle$ dans U , sous la forme

$$h_b(\langle a_0, a_1, \dots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

et soit $\mathcal{H} = \{h_b : b \in \mathbf{Z}_p\}$. Montrer que \mathcal{H} est $((n-1)/p)$ -universelle d'après la définition de ε -universelle de l'exercice 11.3.5. (Conseil : Voir exercice 31.4.4.)

11.4 ADRESSAGE OUVERT

Dans l'*adressage ouvert*, tous les éléments sont conservés dans la table de hachage elle-même. En d'autres termes, chaque entrée de la table contient soit un élément de l'ensemble dynamique, soit NIL. Lorsqu'on recherche un élément, on examine systématiquement les alvéoles de la table jusqu'à trouver l'élément souhaité, ou jusqu'à ce qu'on s'aperçoive que l'élément n'appartient pas à la table. Il n'existe ni listes ni éléments conservés hors de la table, comme c'était le cas pour le chaînage. En adressage ouvert, la table de hachage peut donc se remplir entièrement, de telle manière que plus aucune insertion ne soit possible ; le facteur de remplissage α ne peut jamais être supérieur à 1.

Bien sûr, on pourrait conserver les listes chaînées servant au chaînage à l'intérieur de la table de hachage, dans les alvéoles inutilisées (voir exercice 11.2.4), mais l'avantage de l'adressage ouvert est qu'il évite complètement le recours aux pointeurs. Au lieu de suivre des pointeurs, on *calcule* la séquence d'alvéoles à examiner. La mémoire supplémentaire libérée par la non conservation des pointeurs permet d'augmenter le nombre d'alvéoles de la table pour la même quantité de mémoire, ce qui engendre potentiellement moins de collisions et des recherches plus rapides.

Pour effectuer une insertion à l'aide de l'adressage ouvert, on examine Successivement, on *sonde* comme on dit, la table de hachage jusqu'à ce qu'on trouve une alvéole vide dans laquelle placer la clé. Au lieu de suivre l'ordre $0, 1, \dots, m-1$ (qui demande un temps de recherche $\Theta(n)$), la séquence des positions sondées *dépend de la clé à insérer*. Pour déterminer les alvéoles à sonder, on étend la fonction de hachage pour y inclure le nombre de sondages (en partant de 0) comme second argument. Ainsi, la fonction de hachage devient

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\} .$$

Avec l'adressage ouvert, il faut que pour chaque clé k , la *séquence de sondage*

$$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

soit une permutation de $\langle 0, 1, \dots, m-1 \rangle$, de façon que chaque position de la table de hachage finisse par être considérée comme une alvéole pour une nouvelle clé lors du remplissage de la table. Dans le pseudo-code suivant, on suppose que les éléments de la table de hachage T sont des clés dépourvues d'informations satellites ; la clé k est identique à l'élément contenant la clé k . Chaque alvéole contient soit une clé, soit NIL (si l'alvéole est vide).

INSÉRER-HACHAGE(T, k)

```

1   $i \leftarrow 0$ 
2  répéter  $j \leftarrow h(k, i)$ 
3      si  $T[j] = \text{NIL}$ 
4          alors  $T[j] \leftarrow k$ 
5          retourner  $j$ 
6      sinon  $i \leftarrow i + 1$ 
7  jusqu'à  $i = m$ 
8  erreur « débordement de la table de hachage »

```

L'algorithme de recherche d'une clé k sonde la même séquence d'alvéoles que l'algorithme d'insertion étudié lorsqu'il fallait insérer la clé k . Du coup, la recherche peut se terminer (par un échec) lorsque elle rencontre une alvéole vide, puisque k aurait été inséré ici et pas plus loin dans la séquence de sondage. (Cette démonstration suppose que les clés ne sont pas supprimées de la table de hachage.) La procédure RECHERCHER-HACHAGE prend en entrée une table de hachage T et une clé k et elle retourne j s'il s'avère que l'alvéole j contient la clé k , ou NIL si la clé k n'est pas présente dans la table T .

RECHERCHER-HACHAGE(T, k)

```

1   $i \leftarrow 0$ 
2  répéter  $j \leftarrow h(k, i)$ 
3      si  $T[j] = k$ 
4          alors retourner  $j$ 
5           $i \leftarrow i + 1$ 
6  jusqu'à  $T[j] = \text{NIL}$  ou  $i = m$ 
7  retourner NIL

```

La suppression dans une table à adressage ouvert est difficile. Quand on supprime une clé dans l'alvéole i , on ne peut pas se contenter de marquer l'alvéole comme étant vide en y plaçant NIL : cela pourrait rendre impossible l'accès à une clé k durant l'insertion de laquelle on aurait sondé l'alvéole i et trouvé qu'elle était occupée. Une solution est de marquer l'alvéole en y stockant la valeur spéciale SUPPRIMÉE au lieu de NIL. Nous pourrions alors modifier la procédure INSÉRER-HACHAGE pour traiter ce type d'alvéoles comme s'il s'agissait d'alvéoles vides susceptibles de recevoir de nouvelles clé. RECHERCHER-HACHAGE ne nécessite aucune modification, puisque les valeurs SUPPRIMÉE sont sautées lors de la recherche. Toutefois, lorsqu'on procède de cette manière, les temps de recherche ne dépendent plus du facteur de remplissage α et on préfère en général choisir la technique de chaînage pour résoudre les collisions lorsque des clés doivent être supprimées.

Dans notre analyse, nous faisons l'hypothèse d'un *hachage uniforme* : on suppose que chacune des $m!$ permutations possibles de $\{0, 1, \dots, m - 1\}$ a autant de chances de constituer la séquence de sondage de chaque clé. Le hachage uniforme généralise la notion de hachage uniforme simple définie précédemment aux contextes où

la fonction de hachage ne produit pas un nombre unique, mais une séquence de sondage complète. Cela dit, le véritable hachage uniforme est difficile à implémenter et, en pratique, on utilisera des approximations pertinentes (comme le double hachage, défini plus loin).

Trois techniques sont souvent utilisées pour calculer les séquences de sondage requises pour l'adressage ouvert : le sondage linéaire, le sondage quadratique et le double hachage. Ces techniques garantissent toutes que $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ est une permutation de $\langle 0, 1, \dots, m-1 \rangle$ pour chaque clé k . Toutefois, aucune de ces techniques ne vérifie entièrement l'hypothèse du hachage uniforme, car aucune d'elles n'est capable de générer plus de m^2 séquences de sondage différentes (au lieu des $m!$ requises par le hachage uniforme). Le double hachage détient le plus grand nombre de séquences de sondage et, comme on pouvait s'y attendre, semble donner les meilleurs résultats.

a) Sondage linéaire

Étant donnée une fonction de hachage ordinaire $h' : U \rightarrow \{0, 1, \dots, m-1\}$, appelée ici **fonction de hachage auxiliaire**, la méthode du **sondage linéaire** utilise la fonction de hachage

$$h(k, i) = (h'(k) + i) \bmod m$$

pour $i = 0, 1, \dots, m-1$. Étant donnée une clé k , la première alvéole sondée est $T[h'(k)]$, c'est-à-dire l'alvéole donnée par la fonction de hachage auxiliaire. On sonde ensuite l'alvéole $T[h'(k) + 1]$ et ainsi de suite jusqu'à l'alvéole $T[m-1]$. Puis, on revient aux alvéoles $T[0], T[1], \dots$, jusqu'à sonder finalement l'alvéole $T[h'(k) - 1]$. Comme le sondage initial détermine la séquence de sondage toute entière, le sondage linéaire n'utilise que m séquences de sondage distinctes.

Le sondage linéaire est facile à implémenter, mais il souffre d'un problème dit **grappe première**. De longues suites d'alvéoles occupées se créent, augmentant ainsi le temps de recherche moyen. Il se forme des grappes, puisqu'une alvéole vide précédée de i alvéoles pleines sera la prochaine à être remplie avec une probabilité de $(i+1)/m$. Les suites d'alvéoles occupées tendent à s'allonger et le temps moyen de recherche à augmenter.

b) Sondage quadratique

Le **sondage quadratique** fait appel à une fonction de hachage de la forme

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \quad (11.5)$$

où h' est une fonction de hachage auxiliaire, c_1 et $c_2 \neq 0$ sont des constantes auxiliaires et $i = 0, 1, \dots, m-1$. La première position sondée est $T[h'(k)]$; les positions suivantes sont décalées selon des valeurs qui forment une fonction quadratique du numéro de sondage i . Cette méthode fonctionne beaucoup mieux que le sondage linéaire, mais pour utiliser complètement la table de hachage, les valeurs de c_1, c_2 et

m sont imposées. (Le problème 11.3 montrera une façon de choisir ces paramètres). De plus, si la première position de sondage est la même pour deux clés quelconques, alors leurs séquences de sondage sont les mêmes, puisque $h(k_1, 0) = h(k_2, 0)$ implique $h(k_1, i) = h(k_2, i)$. Cela conduit à une forme bénigne de la maladie de la grappe, appelée *grappe secondaire*. Comme pour le sondage linéaire, le sondage initial détermine la séquence toute entière, de sorte que l'on n'a besoin que de m séquences de sondage distinctes.

c) Double hachage

Le double hachage est l'une des meilleures méthodes connues pour l'adressage ouvert, car les permutations générées ont nombre des caractéristiques des permutations choisies aléatoirement. Le *double hachage* utilise une fonction de hachage de la forme

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

où h_1 et h_2 sont des fonctions de hachage auxiliaires. La position initiale de sondage est $T[h_1(k)]$; les positions de sondage suivantes s'obtiennent par décalage à partir des positions précédentes, décalage valant $h_2(k)$ modulo m . Ainsi, contrairement au cas des sondages linéaire ou quadratique, la séquence de sondage dépend ici de deux manières de la clé k , puisque la position initiale, le décalage, ou les deux, peuvent varier. La figure 11.5 donne un exemple d'insertion par double hachage.

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Figure 11.5 Insertion par double hachage. Nous avons ici une table de hachage de taille 13 avec $h_1(k) = k \bmod 13$ et $h_2(k) = 1 + (k \bmod 11)$. Puisque $14 \equiv 1 \bmod 13$ et $14 \equiv 3 \bmod 11$, la clé 14 sera insérée dans l'alvéole vide 9, après que les alvéoles 1 et 5 auront été examinées et trouvées occupées.

Le nombre $h_2(k)$ doit être premier avec la taille m de la table de hachage pour que la table soit parcourue entièrement. (Voir exercice 11.4-3.) Un moyen commode de

garantir cette condition est de prendre pour m une puissance de 2 et de concevoir h_2 pour qu'elle produise toujours un nombre impair. Une autre possibilité est de choisir m premier et de concevoir h_2 pour qu'elle retourne toujours un entier positif inférieur à m . Par exemple, on pourrait prendre m premier et avoir

$$\begin{aligned}h_1(k) &= k \bmod m, \\h_2(k) &= 1 + (k \bmod m'),\end{aligned}$$

où m' est pris légèrement inférieur à m (disons, $m - 1$). Par exemple, si $k = 123456$, $m = 701$ et $m' = 700$, on a $h_1(k) = 80$ et $h_2(k) = 257$, ce qui donne 80 comme première position de sondage et permet ensuite d'examiner une alvéole sur 257 (modulo m) jusqu'à ce que la clé soit trouvée ou que chaque alvéole ait été examinée.

Le double hachage représente une amélioration par rapport aux sondages linéaire et quadratique, au sens où $\Theta(m^2)$ séquences de sondage sont utilisées au lieu de $\Theta(m)$, puisque chaque paire $(h_1(k), h_2(k))$ possible engendre une séquence de sondage distincte. Ainsi, les performances du double hachage semblent très proches de celle du schéma « idéal » du hachage uniforme.

d) Analyse de l'adressage ouvert

Notre analyse de l'adressage ouvert, comme celle du chaînage, s'exprime en fonction du facteur de remplissage $\alpha = n/m$ de la table de hachage, quand n et m tendent vers l'infini. Bien sûr, avec l'adressage ouvert, on a au plus un élément par alvéole et donc $n \leq m$, ce qui implique $\alpha \leq 1$.

On suppose qu'on utilise un hachage uniforme. Dans ce schéma idéalisé, la séquence de sondage $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ employée pour insérer ou rechercher chaque clé k la même probabilité d'être l'une quelconque des permutations de $\langle 0, 1, \dots, m - 1 \rangle$. Bien sûr, une clé donnée est associée à une séquence de sondage unique ; ce qu'on entend par là est que, si l'on considère la distribution de probabilité sur l'espace des clés et l'action de la fonction de hachage sur les clés, toutes les séquences de sondage possibles sont équiprobables.

Nous allons maintenant analyser le nombre attendu de sondages pour le hachage par adressage ouvert dans l'hypothèse d'un hachage uniforme, en commençant par une analyse du nombre de sondages effectués lors d'une recherche infructueuse.

Théorème 11.6 *Étant donnée une table de hachage pour adressage ouvert avec un facteur de remplissage $\alpha = n/m < 1$, le nombre moyen de sondages pour une recherche infructueuse vaut au plus $1/(1 - \alpha)$, si l'on suppose que le hachage est uniforme.*

Démonstration : Lors d'une recherche infructueuse, chaque sondage hormis le dernier accède à une alvéole occupée qui ne contient pas la clé voulue et la dernière alvéole sondée est vide. Définissons la variable aléatoire X comme étant le nombre de sondages faits au cours d'une recherche infructueuse, et définissons l'événement A_i , pour $i = 1, 2, \dots$, comme étant l'événement dans lequel il y a un i ème

sondage qui détecte une alvéole occupée. Alors, l'événement $\{X \geq i\}$ est l'intersection des événements $A_1 \cap A_2 \cap \dots \cap A_{i-1}$. Nous allons borner $\Pr\{X \geq i\}$ en bornant $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$. D'après l'exercice C.26,

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \dots \cap A_{i-2}\}.$$

Comme il y a n éléments et m alvéoles, $\Pr\{A_1\} = n/m$. Pour $j > 1$, la probabilité qu'il y ait un j ème sondage qui détecte une alvéole occupée, sachant que les $j - 1$ premiers sondages avaient détecté des alvéoles occupées, est $(n - j + 1)/(m - j + 1)$. Cette probabilité s'explique ainsi : on est en train de trouver l'un des $(n - (j - 1))$ éléments restants dans l'une des $(m - (j - 1))$ alvéoles non examinées ; or, d'après l'hypothèse de hachage uniforme, la probabilité est le rapport de ces quantités. En observant que $n < m$ entraîne que $(n - j)/(m - j) \leq n/m$ pour tout j tel que $0 \leq j < m$, on a pour tout i tel que $1 \leq i \leq m$,

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ \leq \left(\frac{n}{m}\right)^{i-1} \\ = \alpha^{i-1}.$$

Nous utilisons maintenant l'équation (C.24) pour borner le nombre attendu de sondages :

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ \leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ = \sum_{i=0}^{\infty} \alpha^i \\ = \frac{1}{1-\alpha}.$$

□

La borne précédente $1 + \alpha + \alpha^2 + \alpha^3 + \dots$ a une interprétation intuitive. Il y a toujours au moins un sondage. Avec une probabilité d'environ α , le premier sondage trouve une alvéole occupée, ce qui justifie alors un second sondage. Avec une probabilité d'environ α^2 , les deux premières alvéoles sondées sont occupées, de sorte qu'il faut un troisième sondage, etc.

Si α est une constante, le théorème 11.6 prédit qu'une recherche infructueuse s'exécute en temps $O(1)$. Par exemple, si la table de hachage est à moitié pleine, le nombre moyen de sondages d'une recherche infructueuse est au plus $1/(1 - 0,5) = 2$. Si elle est remplie à 90%, le nombre moyen de sondages vaut au plus $1/(1 - 0,9) = 10$.

Le théorème 11.6 donne les performances de la procédure INSÉRER-HACHAGE presque immédiatement.

Corollaire 11.7 *Insérer un élément dans une table de hachage pour adressage ouvert dotée d'un facteur de remplissage α demande au plus $1/(1-\alpha)$ sondages en moyenne, dans l'hypothèse où il y a hachage uniforme.*

Démonstration : Un élément n'est inséré que s'il y a de la place dans la table, et donc que si $\alpha < 1$. L'insertion d'une clé équivaut à une recherche infructueuse, suivie du placement de la clé dans la première alvéole trouvée. Donc, le nombre attendu de sondages vaut au plus $1/(1-\alpha)$. \square

Le calcul du nombre attendu de sondages pour une recherche réussie demande un peu plus de travail.

Théorème 11.8 *Étant donnée une table de hachage pour adressage ouvert ayant un facteur de remplissage $\alpha < 1$, le nombre moyen de sondages pour une recherche fructueuse vaut au plus*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

en supposant que le hachage est uniforme et que chaque clé de la table a autant de chance d'être recherchée que les autres.

Démonstration : La recherche d'une clé k suit la même séquence de sondage que celle utilisée pour insérer l'élément de clé k . D'après le corollaire 11.7, si k était la $(i+1)$ ième clé insérée dans la table de hachage, le nombre attendu de sondages effectués lors d'une recherche de la clé k vaut au plus $1/(1-i/m) = m/(m-i)$. Si l'on fait la moyenne sur les n clés de la table de hachage, on obtient le nombre moyen de sondages lors d'une recherche fructueuse :

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha} (H_m - H_{m-n}), \end{aligned}$$

où $H_i = \sum_{j=1}^i 1/j$ est le i ème nombre harmonique (tel que défini dans l'équation (A.7)). En utilisant la technique décrite à la section A.2 consistant à borner une sommation par une intégrale, on obtient

$$\begin{aligned} \frac{1}{\alpha} (H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^m 1/k \\ &\leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{d'après l'inégalité (A.12)}) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

comme borne pour le nombre attendu de sondages lors d'une recherche fructueuse. \square

Si la table de hachage est à moitié remplie, le nombre attendu de sondages lors d'une recherche fructueuse est inférieur à 1,387. Si la table de hachage est remplie à 90 %, ce nombre moyen est inférieur à 2,559.

Exercices

11.4.1 On considère l'insertion des clés 10, 22, 31, 4, 15, 28, 17, 88, 59 dans une table de hachage de longueur $m = 11$ en utilisant l'adressage ouvert avec pour fonction de hachage auxiliaire $h'(k) = k \bmod m$. Illustrer le résultat de l'insertion de ces clés par sondage linéaire, par sondage quadratique avec $c_1 = 1$ et $c_2 = 3$, et par double hachage avec $h_2(k) = 1 + (k \bmod (m - 1))$.

11.4.2 Écrire un pseudo code pour la procédure SUPPRIMER-HACHAGE telle que définie dans le texte, et modifier INSÉRER-HACHAGE pour qu'elle gère la valeur spéciale SUPPRIMÉE.

11.4.3 * Supposons que l'on utilise le double hachage pour gérer les collisions ; autrement dit, on utilise la fonction de hachage $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Montrer que, si m et $h_2(k)$ ont un plus grand commun diviseur $d \geq 1$ pour une certaine clé k , alors une recherche infructueuse de la clé k examine $(1/d) \%$ de la table de hachage avant de revenir à l'alvéole $h_1(k)$. Donc, quand $d = 1$, auquel cas m et $h_2(k)$ sont premiers entre eux, la recherche risque de balayer toute la table de hachage. (*Conseil* : Voir chapitre 31).

11.4.4 On considère une table de hachage à adressage ouvert avec hachage uniforme. Donner des majorants pour le nombre moyen de sondages par recherche infructueuse et pour le nombre attendu de sondages par recherche fructueuse, pour un facteur de remplissage égal à $3/4$ puis à $7/8$.

11.4.5 * On considère une table de hachage à adressage ouvert ayant un facteur de remplissage α . Trouver la valeur non nulle α pour laquelle le nombre moyen de sondages dans une recherche infructueuse vaut deux fois le nombre moyen de sondages dans une recherche fructueuse. Utiliser les majorants donnés par les théorèmes 11.6 et 11.8 concernant ces nombres moyens de sondages.

11.5^{*} HACHAGE PARFAIT

Le hachage s'utilise le plus souvent pour ses excellentes performances moyennes, mais il donne aussi des performances excellentes dans le cas le plus défavorable quand l'ensemble des clés est *statique* : une fois les clés stockées dans la table, l'ensemble des clés ne change plus. Certaines applications gèrent des ensembles statiques de clés : ensemble des mots réservés d'un langage de programmation ; ensemble des noms de fichier sur un CD-ROM ; etc. On dit d'une technique de hachage que c'est un *hachage parfait* si le nombre d'accès mémoire requis pour faire une recherche est, dans le cas le plus défavorable, $O(1)$.

L'idée fondamentale pour créer un mécanisme de hachage parfait est simple. On utilise une stratégie de hachage à deux niveaux, avec un hachage universel à chaque niveau. La figure 11.6 illustre cette démarche.

Le premier niveau est essentiellement le même que pour le hachage avec chaînage : les n clés sont hachées vers m alvéoles, à l'aide d'une fonction de hachage h soigneusement sélectionnée dans une famille de fonctions de hachage universelles.

Mais au lieu de créer une liste des clés dont le hachage a abouti à l'alvéole j , on utilise une petite **table de hachage secondaire** S_j ayant une fonction de hachage associée h_j . En choisissant avec soin les fonctions de hachage h_j , on peut garantir qu'il n'y aura pas de collisions au niveau secondaire.

Pour assurer qu'il n'y aura pas de collisions au niveau secondaire, nous devons cependant faire en sorte que la taille m_j de la table de hachage S_j soit le carré du nombre n_j de clés hachées vers l'alvéole j . Ce genre de dépendance quadratique entre m_j et n_j peut sembler de nature à engendrer des besoins excessifs en matière de capacité globale de stockage ; mais nous montrerons que, si l'on choisit bien la fonction de hachage du premier niveau, la quantité totale attendue d'espace est encore $O(n)$.

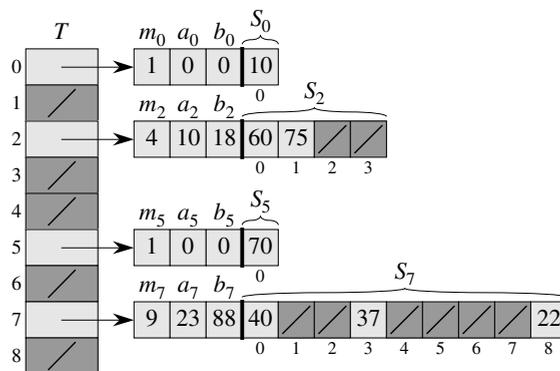


Figure 11.6 Utilisation du hachage parfait pour stocker l'ensemble $K = \{10, 22, 37, 40, 60, 70, 75\}$. La fonction de hachage extérieure est $h(k) = ((ak + b) \bmod p) \bmod m$, avec $a = 3$, $b = 42$, $p = 101$ et $m = 9$. Par exemple, $h(75) = 2$, et donc la clé 75 est hachée vers l'alvéole 2 de la table T . Une table de hachage secondaire S_j stocke toutes les clés hachées vers l'alvéole j . La taille de la table de hachage S_j est m_j , et la fonction de hachage associée est $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Comme $h_2(75) = 1$, la clé 75 est stockée dans l'alvéole 1 de la table de hachage secondaire S_2 . Comme il n'y a pas de collisions dans aucune des tables de hachage secondaires, la recherche prend un temps constant dans le cas le plus défavorable.

Nous utilisons des fonctions de hachage prises dans les classes universelles de fonctions de hachage de la section 11.3.3. La fonction de hachage de premier niveau est choisie dans la classe $\mathcal{H}_{p,m}$ où, comme à la section 11.3.3, p est un nombre premier plus grand que toutes les valeurs de clé. Les clés hachées vers l'alvéole j sont

rehachées vers une table de hachage secondaire S_j de taille m_j , via une fonction de hachage h_j choisie dans la classe \mathcal{H}_{p,m_j} .⁽¹⁾

Nous allons procéder en deux temps. Primo, nous allons voir comment garantir que les tables secondaires n'aient pas de collisions. Secundo, nous allons montrer que la quantité attendue de mémoire utilisée *en tout* (table de hachage principale, plus toutes les tables de hachage secondaires) est $O(n)$.

Théorème 11.9 *Si l'on stocke n clés dans une table de hachage de taille $m = n^2$ via une fonction de hachage h choisie aléatoirement dans une classe universelle de fonctions de hachage, alors la probabilité d'avoir des collisions est inférieure à $1/2$.*

Démonstration : Il y a $\binom{n}{2}$ paires de clés susceptibles d'entrer en collision ; chaque paire présente une probabilité de collision de $1/m$ si h est choisie aléatoirement dans une famille universelle \mathcal{H} de fonctions de hachage. Soit X une variable aléatoire qui compte le nombre de collisions. Quand $m = n^2$, le nombre attendu de collisions est

$$\begin{aligned} E[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2. \end{aligned}$$

(Notez que cette analyse ressemble à celle du paradoxe des anniversaires, vu à la section 5.4.1.) En appliquant l'inégalité de Markov (C.29), $\Pr\{X \geq t\} \leq E[X]/t$, avec $t = 1$ on termine la démonstration. \square

Dans la situation décrite dans le théorème 11.9, où $m = n^2$, il s'ensuit qu'une fonction de hachage h choisie aléatoirement dans \mathcal{H} a plus de chances de ne *pas* avoir de collisions que d'en avoir. Étant donné l'ensemble K des n clés à hacher (rappelez-vous que K est statique), il est donc facile de trouver une fonction de hachage h sans collision avec quelques essais aléatoires.

Cependant, quand n est grand, une table de hachage de taille $m = n^2$ est excessive. On adopte donc une stratégie de hachage à deux niveaux, et l'on utilise la démarche du théorème 11.9 uniquement pour hacher les éléments au sein de chaque alvéole. Une fonction de hachage h extérieure (de premier niveau) sert à hacher les clés vers $m = n$ alvéoles. Ensuite, si n_j clés sont hachées vers l'alvéole j , une table de hachage secondaire S_j de taille $m_j = n_j^2$ permet de faire des consultations en temps constant et sans risque de collision.

Attaquons-nous maintenant au problème de garantir que la mémoire globalement utilisée est $O(n)$. Comme la taille m_j de la j ème table de hachage secondaire croît de façon quadratique avec le nombre n_j de clés stockées, il y a risque que la quantité totale de mémoire soit excessive.

(1) Quand $n_j = m_j = 1$, on n'a pas vraiment besoin de fonction de hachage pour l'alvéole j ; quand on choisit une fonction de hachage $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ pour une telle alvéole, on se contente de faire $a = b = 0$.

Si la taille de la table de premier niveau est $m = n$, alors la quantité de mémoire consommée est $O(n)$ pour la table de hachage principale, pour le stockage des tailles m_j des tables de hachage secondaires et pour le stockage des paramètres a_j et b_j définissant les fonctions de hachage secondaires h_j extraites de la classe \mathcal{H}_{p,m_j} de la section 11.3.3 (sauf quand $n_j = 1$ et que l'on fait $a = b = 0$). Le théorème suivant et un corollaire fournissent une borne pour les tailles combinées attendues de toutes les tables de hachage secondaires. Un second corollaire borne la probabilité que la taille combinée de toutes les tables de hachage secondaires soit supra linéaire.

Théorème 11.10 *Si on stocke n clés dans une table de hachage de taille $m = n$ via une fonction de hachage h choisie aléatoirement dans une classe universelle de fonctions de hachage, alors*

$$\mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] < 2n ,$$

où n_j est le nombre de clés hachées vers l'alvéole j .

Démonstration : Nous commencerons par l'identité suivante, vérifiée pour tout entier non négatif a :

$$a^2 = a + 2 \binom{a}{2} . \quad (11.6)$$

On a

$$\begin{aligned} \mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] &= \mathbb{E} \left[\sum_{j=0}^{m-1} \left(n_j + 2 \binom{n_j}{2} \right) \right] && \text{(d'après l'équation (11.6))} \\ &= \mathbb{E} \left[\sum_{j=0}^{m-1} n_j \right] + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(d'après la linéarité de l'espérance)} \\ &= \mathbb{E} [n] + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(d'après l'équation (11.1))} \\ &= n + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(car } n \text{ n'est pas une variable aléatoire) .} \end{aligned}$$

Pour évaluer la somme $\sum_{j=0}^{m-1} \binom{n_j}{2}$, observons que c'est en fait le nombre total de collisions. D'après les propriétés du hachage universel, la valeur attendue de cette somme est au plus

$$\binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2} ,$$

car $m = n$. D'où,

$$E \left[\sum_{j=0}^{m-1} n_j^2 \right] \leq n + 2 \frac{n-1}{2} = 2n - 1 < 2n .$$

□

Corollaire 11.11 Si l'on stocke n clés dans une table de hachage de taille $m = n$, via une fonction de hachage h choisie aléatoirement dans une classe universelle de fonctions de hachage, et si l'on prend pour chaque table de hachage secondaire une taille $m_j = n_j^2$ (pour $j = 0, 1, \dots, m-1$), alors la quantité moyenne de mémoire requise par toutes les tables de hachage secondaires d'une stratégie de hachage parfait est inférieure à $2n$.

Démonstration : Puisque $m_j = n_j^2$ pour $j = 0, 1, \dots, m-1$, le théorème 11.10 donne

$$E \left[\sum_{j=0}^{m-1} m_j \right] = E \left[\sum_{j=0}^{m-1} n_j^2 \right] < 2n , \quad (11.7)$$

ce qui termine la démonstration. □

Corollaire 11.12 Si l'on stocke n clés dans une table de hachage de taille $m = n$, via une fonction de hachage h choisie aléatoirement dans une classe universelle de fonctions de hachage, et si l'on prend pour chaque table de hachage secondaire une taille $m_j = n_j^2$ (pour $j = 0, 1, \dots, m-1$), alors la probabilité que l'espace total consommé pour les tables de hachage secondaires dépasse $4n$ est inférieure à $1/2$.

Démonstration : On applique derechef l'inégalité de Markov (C.29), $\Pr \{X \geq t\} \leq E[X]/t$, cette fois pour l'inégalité (11.7), avec $X = \sum_{j=0}^{m-1} m_j$ et $t = 4n$:

$$\Pr \left\{ \sum_{j=0}^{m-1} m_j \geq 4n \right\} \leq \frac{E \left[\sum_{j=0}^{m-1} m_j \right]}{4n} < \frac{2n}{4n} = 1/2 .$$

□

D'après le corollaire 11.12, on voit que, si l'on teste un petit nombre de fonctions de hachage choisies aléatoirement dans la famille universelle, on obtient très vite une fonction qui consomme un volume raisonnable de mémoire.

Exercices

11.5.1 * Supposons que l'on insère n clés dans une table de hachage de taille m , via adressage ouvert et hachage uniforme. Soit $p(n, m)$ la probabilité qu'il n'y ait pas de collisions. Montrer que $p(n, m) \leq e^{-n(n-1)/2m}$. (Conseil : Voir équation (3.11).) Prouver que, quand n dépasse \sqrt{m} , la probabilité d'éviter les collisions tend rapidement vers zéro.

PROBLÈMES

11.1. Borne de plus long sondage pour hachage

Une table de hachage de taille m sert à stocker n éléments, avec $n \leq m/2$. Les collisions sont résolues par adressage ouvert.

- a. En faisant l'hypothèse d'un hachage uniforme, montrer que pour $i = 1, 2, \dots, n$, la probabilité pour que la i ème insertion nécessite plus de k sondages vaut au plus 2^{-k} .
- b. Montrer que, pour $i = 1, 2, \dots, n$, la probabilité pour que la i ème insertion nécessite plus de $2 \lg n$ sondages est au plus $1/n^2$.

Soit la variable aléatoire X_i représentant le nombre de sondages requis par la i ème insertion. On a montré dans la partie (b) que $\Pr \{X_i > 2 \lg n\} \leq 1/n^2$. Soit la variable aléatoire $X = \max_{1 \leq i \leq n} X_i$ représentant le nombre maximal de sondages requis par l'une quelconque des n insertions.

- c. Montrer que $\Pr \{X > 2 \lg n\} \leq 1/n$.
- d. Montrer que la longueur attendue $E[X]$ de la séquence de sondage la plus longue est $O(\lg n)$.

11.2. Borne de la taille d'une alvéole pour chaînage

On suppose qu'on a une table de hachage avec n alvéoles et que les collisions sont résolues par chaînage, et on suppose que n clés sont insérées dans la table. Chaque clé a des chances égales d'être hachée vers chaque alvéole. Soit M le nombre maximal de clés dans une alvéole quelconque, après que toutes les clés ont été insérées. Votre mission est de prouver un majorant $O(\lg n / \lg \lg n)$ sur $E[M]$, espérance de M .

- a. Montrer que la probabilité Q_k pour que k clés exactement soient hachées vers une alvéole particulière est donnée par la formule

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

- b. Soit P_k la probabilité pour que $M = k$, c'est-à-dire pour que l'alvéole contenant le plus de clés en contienne k . Montrer que $P_k \leq nQ_k$.
- c. Utiliser la formule de Stirling (équation (3.17)) pour montrer que $Q_k < e^k/k^k$.
- d. Si l'on définit $k_0 = c \lg n / \lg \lg n$, montrer qu'il existe une constante $c > 1$ telle que $Q_{k_0} < 1/n^3$. En déduire que $P_k < 1/n^2$ pour $k \geq k_0 = c \lg n / \lg \lg n$.
- e. Montrer que

$$E[M] \leq \Pr \left\{ M > \frac{c \lg n}{\lg \lg n} \right\} \cdot n + \Pr \left\{ M \leq \frac{c \lg n}{\lg \lg n} \right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

En déduire que $E[M] = O(\lg n / \lg \lg n)$.

11.3. Sondage quadratique

Supposons qu'on doive rechercher une clé k dans une table de hachage comportant les emplacements $0, 1, \dots, m - 1$ et qu'on dispose d'une fonction de hachage h établissant une correspondance entre l'espace des clés et l'ensemble $\{0, 1, \dots, m - 1\}$. Le schéma de recherche est le suivant.

- 1) Calcul de la valeur $i \leftarrow h(k)$ et initialisation de $j \leftarrow 0$.
- 2) Sondage de la position i pour la clé k désirée. En cas de succès, ou si la position est vide, stopper la recherche.
- 3) Initialisation de $j \leftarrow (j + 1) \bmod m$ et $i \leftarrow (i + j) \bmod m$, et retourner à l'étape 2.

On suppose que m est une puissance de 2.

- a. Montrer que ce schéma est un cas particulier du schéma général du « sondage quadratique », en exhibant les constantes c_1 et c_2 idoines pour l'équation (11.5).
- b. Démontrer que, dans le cas le plus défavorable, cet algorithme examine chaque position de la table.

11.4. Hachage k -universel et authentification

Soit $\mathcal{H} = \{h\}$ une classe de fonctions de hachage dans laquelle chaque fonction h établit une correspondance entre l'univers U des clés et $\{0, 1, \dots, m - 1\}$. On dit que \mathcal{H} est **k -universelle** si, pour toute séquence fixée de k clés distinctes $\langle x_1, x_2, \dots, x_k \rangle$ et pour toute fonction h choisie au hasard dans \mathcal{H} , la séquence $\langle h(x_1), h(x_2), \dots, h(x_k) \rangle$ a des chances égales d'être l'une quelconque des m^k séquences de longueur k d'éléments pris dans $\{0, 1, \dots, m - 1\}$.

- a. Montrer que, si \mathcal{H} est 2-universelle, alors elle est universelle.
- b. Soit U l'ensemble des n -uplets de valeurs prises dans \mathbf{Z}_p , et soit $B = \mathbf{Z}_p$, où p est premier. Pour tout n -uplet $a = \langle a_0, a_1, \dots, a_{n-1} \rangle$ de valeurs de \mathbf{Z}_p et pour tout $b \in \mathbf{Z}_p$, on définit la fonction de hachage $h_{a,b} : U \rightarrow B$ qui, à un n -uplet $x = \langle x_0, x_1, \dots, x_{n-1} \rangle$ de U , associe

$$h_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

et soit $\mathcal{H} = \{h_{a,b}\}$. Montrer que \mathcal{H} est 2-universelle.

- c. Alice et Bob se sont secrètement entendus pour choisir une fonction de hachage $h_{a,b}$ dans une famille 2-universelle \mathcal{H} de fonctions de hachage. Plus tard, Alice envoie un message m à Bob via Internet, avec $m \in U$. Elle authentifie ce message pour Bob en envoyant également une balise d'authentification $t = h_{a,b}(m)$, et Bob vérifie que la paire (m, t) qu'il reçoit satisfait à $t = h_{a,b}(m)$. Supposez qu'un espion

intercepte (m, t) en route et essaie de tromper Bob en remplaçant la paire par une autre paire (m', t') . Montrer que la probabilité pour que l'espion réussisse à duper Bob en lui faisant accepter (m', t') est au plus $1/p$, quelle que soit la puissance de traitement informatique dont dispose l'espion.

NOTES

Knuth [185] et Gonnet [126] sont d'excellentes références pour l'analyse des algorithmes de hachage. Knuth attribue à H. P. Luhn (1953) l'invention des tables de hachage, ainsi que la méthode de résolution des collisions par chaînage. À peu près à la même époque, G. M. Amdahl jeta les bases de l'adressage ouvert.

Carter et Wegman ont introduit la notion de classes universelles de fonctions de hachage en 1979 [52].

Fredman, Komlós et Szemerédi [96] ont inventé la stratégie du hachage parfait pour ensembles statiques, étudiée à la section 11.5. Une extension de leur méthode aux ensembles dynamiques, avec gestion des insertions et des suppressions en temps moyen amorti $O(1)$, a été donnée par Dietzfelbinger et d'autres. [73].

Chapitre 12

Arbres binaires de recherche

Les arbres⁽¹⁾ de recherche sont des structures de données pouvant supporter nombre d'opérations d'ensemble dynamique, notamment RECHERCHER, MINIMUM, MAXIMUM, PRÉDÉCESSEUR, SUCESSEUR, INSÉRER et SUPPRIMER. Un arbre de recherche peut donc servir aussi bien de dictionnaire que de file de priorités.

Les opérations basiques sur un arbre binaire de recherche dépensent un temps proportionnel à la hauteur de l'arbre. Pour un arbre binaire complet à n nœuds, ces opérations s'exécutent en $\Theta(\lg n)$ dans le cas le plus défavorable. En revanche, quand l'arbre se réduit à une chaîne linéaire de n nœuds, les mêmes opérations requièrent un temps $\Theta(n)$ dans le cas le plus défavorable. Nous verrons à la section 12.4 que la hauteur attendue d'un arbre binaire de recherche construit aléatoirement est $O(\lg n)$, ce qui fait que les opérations de base d'ensemble dynamique sur un tel arbre requièrent un temps $\Theta(\lg n)$ en moyenne.

En pratique, on ne peut pas toujours assurer que les arbres binaires de recherche sont construits aléatoirement, mais certaines variantes de ces arbres permettent de garantir de bonnes performances dans le cas le plus défavorable pour les opérations fondamentales. Une de ces variantes, les arbres rouge-noir dont la hauteur est $O(\lg n)$, est présentée au chapitre 13. Le chapitre 18 introduit les B-arbres, qui sont particulièrement efficaces pour gérer des bases de données sur des espaces de stockage secondaire (disques) à accès aléatoire.

Après une présentation des propriétés fondamentales des arbres binaires de recherche, les sections suivantes expliquent comment parcourir un tel arbre pour afficher les valeurs en ordre trié, comment y rechercher une valeur particulière, comment

(1) Dans cette section, ainsi qu'aux chapitres 13 et 14, nous emploierons le terme « arbre » par abus de langage, en lieu et place « d'arborescence ». (Voir annexe B .)

trouver l'élément minimal ou maximal, comment trouver le prédécesseur ou le successeur d'un élément et comment insérer ou supprimer un élément. Les propriétés mathématiques fondamentales des arbres sont présentées à l'annexe B.

12.1 QU'EST-CE QU'UN ARBRE BINAIRE DE RECHERCHE ?

Comme son nom l'indique, un arbre binaire de recherche est organisé comme un arbre binaire, ainsi qu'on peut le voir à la figure 12.1. On peut le représenter par une structure de données chaînée dans laquelle chaque nœud est un objet. En plus du champ *clé* et des données satellites, chaque nœud contient les champs *gauche*, *droite* et *p* qui pointent sur les nœuds correspondant respectivement à son enfant de gauche, à son enfant de droite et à son parent. Si le parent, ou l'un des enfant, est absent, le champ correspondant contient la valeur NIL. Le nœud racine est le seul nœud de l'arbre dont le champ parent vaut NIL.

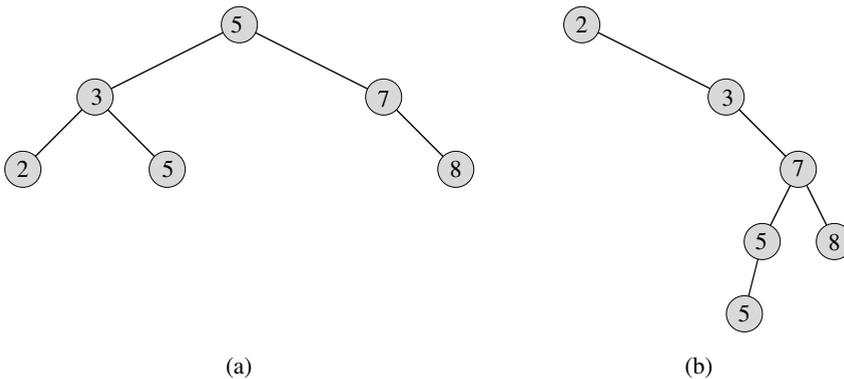


Figure 12.1 Arbres binaires de recherche. Pour un nœud x , les clés du sous-arbre de gauche de x valent au plus $clé[x]$ et celles du sous-arbre de droite valent au moins $clé[x]$. Des arbres binaires de recherche différents peuvent représenter le même ensemble de valeurs. Le temps d'exécution, dans le cas le plus défavorable, pour la plupart des opérations d'arbre de recherche est proportionnel à la hauteur de l'arbre. (a) Un arbre binaire de recherche de 6 nœuds et de hauteur 2. (b) Un arbre binaire de recherche moins efficace de hauteur 4, contenant les mêmes clés.

Les clés d'un arbre binaire de recherche sont toujours stockées de manière à satisfaire à la **propriété d'arbre binaire de recherche** :

Soit x un nœud d'un arbre binaire de recherche. Si y est un nœud du sous-arbre de gauche de x , alors $clé[y] \leq clé[x]$. Si y est un nœud du sous-arbre de droite de x , alors $clé[x] \leq clé[y]$.

Ainsi, dans la figure 12.1(a), la clé de la racine est 5, les clés 2, 3 et 5 de son sous-arbre de gauche sont inférieures ou égales à 5 et les clés 7 et 8 de son sous-arbre de droite sont supérieures ou égales à 5. La propriété reste valable pour chaque nœud de l'arbre. Par exemple, la clé 3 de la figure 12.1(a) n'est pas inférieure à la clé 2 de son sous-arbre de gauche, ni supérieure à la clé 5 de son sous-arbre de droite.

La propriété d'arbre binaire de recherche permet d'afficher toutes les clés de l'arbre dans l'ordre trié à l'aide d'un algorithme récursif simple, appelé *parcours infixe* : la clé de la racine d'un sous-arbre est imprimée entre les clés du sous-arbre de gauche et les clés du sous-arbre de droite. (De même, un *parcours préfixe* imprimera la racine avant les valeurs de chacun de ses sous-arbres et un *parcours postfixe* imprimera la racine après les valeurs de ses sous-arbres.) Pour imprimer tous les éléments d'un arbre binaire de recherche T à l'aide de la procédure suivante, on appelle $\text{PARCOURS-INFIXE}(\text{racine}[T])$.

$\text{PARCOURS-INFIXE}(x)$

- 1 **si** $x \neq \text{NIL}$
- 2 **alors** $\text{PARCOURS-INFIXE}(\text{gauche}[x])$
- 3 afficher $\text{clé}[x]$
- 4 $\text{PARCOURS-INFIXE}(\text{droite}[x])$

Par exemple, le parcours infixe imprimera les clés de chacun des deux arbres de la figure 12.1 dans l'ordre 2, 3, 5, 5, 7, 8. La validité de l'algorithme se déduit directement par récurrence de la propriété d'arbre binaire de recherche. Le parcours d'un arbre binaire de recherche à n nœuds prend un temps $\Theta(n)$ puisque, après le premier appel, la procédure est appelée récursivement exactement deux fois pour chaque nœud de l'arbre (une fois pour son enfant de gauche et une fois pour son enfant de droite). Le théorème suivant donne une démonstration plus formelle du fait qu'il faut un temps linéaire pour faire un parcours infixe de l'arbre.

Théorème 12.1 *Si x est la racine d'un sous-arbre à n nœuds, alors l'appel $\text{PARCOURS-INFIXE}(x)$ prend un temps $\Theta(n)$.*

Démonstration : Soit $T(n)$ le temps que met PARCOURS-INFIXE quand on l'appelle pour la racine d'un sous-arbre à n nœuds. PARCOURS-INFIXE prend une petite quantité de temps constante pour un sous-arbre vide (pour faire le test $x \neq \text{NIL}$), et donc $T(0) = c$ où c est une certaine constante positive.

Pour $n > 0$, supposons que PARCOURS-INFIXE soit appelée sur un nœud x dont le sous-arbre de gauche a k nœuds et dont le sous-arbre de droite a $n - k - 1$ nœuds. Le temps d'exécution de $\text{PARCOURS-INFIXE}(x)$ est $T(n) = T(k) + T(n - k - 1) + d$, d étant une constante positive qui reflète le temps d'exécution de PARCOURS-INFIXE quand on ne tient pas compte du temps consommé dans les appels récursifs.

On va utiliser la méthode de substitution pour montrer que $T(n) = \Theta(n)$ en prouvant que $T(n) = (c + d)n + c$. Pour $n = 0$, on a $(c + d) \cdot 0 + c = c = T(0)$. Pour $n > 0$, on a

$$\begin{aligned}
 T(n) &= T(k) + T(n - k - 1) + d \\
 &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\
 &= (c + d)n + c - (c + d) + c + d \\
 &= (c + d)n + c,
 \end{aligned}$$

ce qui termine la démonstration. □

Exercices

12.1.1 Dessiner des arbres binaires de recherche de hauteur 2, 3, 4, 5 et 6 pour le même ensemble de clés $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1.2 Quelle est la différence entre la propriété d'arbre binaire de recherche et la propriété de tas min (voir page 123) ? Est-ce que la propriété de tas min permet d'afficher en ordre trié et en temps $O(n)$ les clés d'un arbre de n nœuds ? Justifier la réponse.

12.1.3 Donner un algorithme non récursif qui effectue un parcours infixe. (*conseil* : Il existe une solution simple qui fait appel à une pile comme structure de données auxiliaire et une solution plus compliquée, mais plus élégante, qui n'utilise aucune pile mais qui suppose qu'on peut tester l'égalité de deux pointeurs.)

12.1.4 Donner des algorithmes récursifs qui effectuent les parcours préfixe et postfixe en un temps $\Theta(n)$ sur un arbre à n nœuds.

12.1.5 Montrer que, puisque le tri de n éléments prend un temps $\Omega(n \lg n)$ dans le cas le plus défavorable dans le modèle de comparaison, tout algorithme basé sur ce modèle qui construit un arbre binaire de recherche à partir d'une liste arbitraire de n éléments s'effectue au pire en $\Omega(n \lg n)$.

12.2 REQUÊTE DANS UN ARBRE BINAIRE DE RECHERCHE

Une opération très courante sur un arbre binaire de recherche est la recherche d'une clé stockée dans l'arbre. En dehors de l'opération RECHERCHER, les arbres binaires de recherche peuvent supporter des requêtes comme MINIMUM, MAXIMUM, SUCESSEUR et PRÉDÉCESSEUR. Dans cette section, on examinera ces opérations et on montrera que chacune peut s'exécuter dans un temps $O(h)$ sur un arbre binaire de recherche de hauteur h .

a) Recherche

On utilise la procédure suivante pour rechercher un nœud ayant une clé donnée dans un arbre binaire de recherche. Étant donné un pointeur sur la racine de l'arbre et une clé k , ARBRE-RECHERCHER retourne un pointeur sur un nœud de clé k s'il en existe un ; sinon, elle retourne NIL.

ARBRE-RECHERCHER(x, k)

1 **si** $x = \text{NIL}$ ou $k = \text{clé}[x]$

2 **alors retourner** x

3 **si** $k < \text{clé}[x]$

4 **alors retourner** ARBRE-RECHERCHER(*gauche*[x], k)

5 **sinon retourner** ARBRE-RECHERCHER(*droite*[x], k)

La procédure commence sa recherche à la racine et suit un chemin descendant, comme le montre la figure 12.2. Pour chaque nœud x qu'elle rencontre, elle compare la clé k avec $clé[x]$. Si les deux clés sont égales, la recherche est terminée. Si k est plus petite que $clé[x]$, la recherche continue dans le sous-arbre de gauche de x puisque la propriété d'arbre binaire de recherche implique que k ne peut pas se trouver dans le sous-arbre de droite. Symétriquement, si k est plus grande que $clé[x]$, la recherche continue dans le sous-arbre de droite. Les nœuds rencontrés pendant la récursivité forment un chemin descendant issu de la racine de l'arbre, et le temps d'exécution de ARBRE-RECHERCHER est donc $O(h)$ si h est la hauteur de l'arbre.

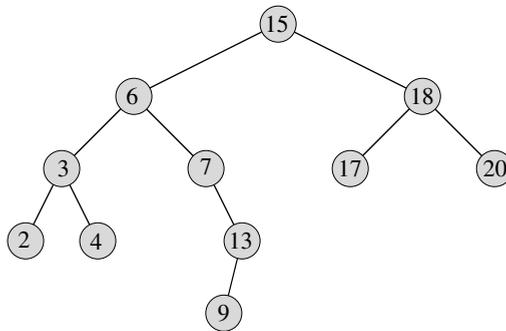


Figure 12.2 Interrogations d'un arbre binaire de recherche. Pour rechercher la clé 13 dans l'arbre, on suit le chemin $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ en partant de la racine. La clé minimale de l'arbre est 2, qu'on peut atteindre en suivant les pointeurs *gauche* à partir de la racine. La clé maximale 20 est trouvée en suivant les pointeurs *droite* à partir de la racine. Le successeur du nœud de clé 15 est le nœud de clé 17, puisque c'est la clé minimale du sous-arbre de droite de 15. Le nœud de clé 13 ne possédant pas de sous-arbre de droite, son successeur est le premier de ses ancêtres (en remontant) dont l'enfant de gauche est aussi un ancêtre. Ici, c'est donc le nœud de clé 15.

La même procédure peut être réécrite de façon itérative, en « déroulant » la récursivité pour en faire une boucle **tant que**. Sur la plupart des ordinateurs, cette version est plus efficace.

ARBRE-RECHERCHER-ITÉRATIF(x, k)

- 1 **tant que** $x \neq \text{NIL}$ et $k \neq clé[x]$
- 2 **faire si** $k < clé[x]$
- 3 **alors** $x \leftarrow gauche[x]$
- 4 **sinon** $x \leftarrow droite[x]$
- 5 **retourner** x

b) Minimum et maximum

On peut toujours trouver un élément d'un arbre binaire de recherche dont la clé est un minimum en suivant les pointeurs *gauche* à partir de la racine jusqu'à ce qu'on

rencontre NIL, comme le montre la figure 12.2. La procédure suivante retourne un pointeur sur l'élément minimal du sous-arbre enraciné au nœud x .

```
ARBRE-MINIMUM( $x$ )
1  tant que gauche[ $x$ ]  $\neq$  NIL
2    faire  $x \leftarrow$  gauche[ $x$ ]
3  retourner  $x$ 
```

La propriété d'arbre binaire de recherche garantit la validité de ARBRE-MINIMUM. Si un nœud x ne possède pas de sous-arbre de gauche, alors comme chaque clé du sous-arbre de droite de x est supérieure ou égale à $clé[x]$, la clé minimale dans le sous-arbre enraciné en x est $clé[x]$. Si le nœud x possède un sous-arbre de gauche, alors comme aucune clé du sous-arbre de droite n'est plus petite que $clé[x]$ et que chaque clé du sous-arbre gauche est inférieure ou égale à $clé[x]$, la clé minimale du sous-arbre enraciné en x se trouve dans le sous-arbre enraciné en gauche[x].

Le pseudo code de ARBRE-MAXIMUM est symétrique.

```
ARBRE-MAXIMUM( $x$ )
1  tant que droite[ $x$ ]  $\neq$  NIL
2    faire  $x \leftarrow$  droite[ $x$ ]
3  retourner  $x$ 
```

Ces deux procédures s'exécutent en $O(h)$ pour un arbre de hauteur h puisque, comme dans ARBRE-RECHERCHER, elles suivent des chemins descendants qui partent de la racine.

c) Successeur et prédécesseur

Étant donné un nœud d'un arbre binaire de recherche, il est parfois utile de pouvoir trouver son successeur dans l'ordre déterminé par un parcours infixe de l'arbre. Si toutes les clés sont distinctes, le successeur d'un nœud x est le nœud possédant la plus petite clé supérieure à $clé[x]$. La structure d'un arbre binaire de recherche permet de déterminer le successeur d'un nœud sans même effectuer de comparaison entre les clés. La procédure suivante retourne le successeur d'un nœud x dans un arbre binaire de recherche, s'il existe et NIL si x possède la plus grande clé de l'arbre.

```
ARBRE-SUCESSEUR( $x$ )
1  si droite[ $x$ ]  $\neq$  NIL
2    alors retourner ARBRE-MINIMUM(droite[ $x$ ])
3   $y \leftarrow p[x]$ 
4  tant que  $y \neq$  NIL et  $x = droite[y]$ 
5    faire  $x \leftarrow y$ 
6     $y \leftarrow p[y]$ 
7  retourner  $y$ 
```

Le code de ARBRE-SUCESSEUR est séparé en deux cas. Si le sous-arbre de droite du nœud x n'est pas vide, alors le successeur de x est tout simplement le nœud le plus à gauche dans le sous-arbre de droite, qui est trouvé en ligne 2 en appelant ARBRE-MINIMUM(*droite*[x]). Par exemple, le successeur du nœud de clé 15 dans la figure 12.2 est le nœud de clé 17.

En revanche, ainsi que l'exercice 12.2.6 vous demandera de le montrer, si le sous-arbre de droite du nœud x est vide et que x a un successeur y , alors y est le premier ancêtre de x dont l'enfant de gauche est aussi un ancêtre de x . Sur la figure 12.2, le successeur du nœud de clé 13 est le nœud de clé 15. Pour trouver y , on remonte tout simplement l'arbre à partir de x jusqu'à trouver un nœud qui soit l'enfant de gauche de son parent ; cette remontée est effectuée par les lignes 3–7 de ARBRE-SUCESSEUR.

Le temps d'exécution de ARBRE-SUCESSEUR sur un arbre de hauteur h est $O(h)$; en effet, on suit soit un chemin vers le haut de l'arbre, soit un chemin vers le bas. La procédure ARBRE-PRÉDÉCESSEUR, qui est symétrique de ARBRE-SUCESSEUR, s'exécute également dans un temps $O(h)$.

Même si les clés ne sont pas distinctes, on définit le successeur et le prédécesseur d'un nœud x comme étant le nœud retourné par des appels à ARBRE-SUCESSEUR(x) et ARBRE-PRÉDÉCESSEUR(x) respectivement.

En résumé, nous venons de démontrer le théorème suivant :

Théorème 12.2 *Les opérations d'ensemble dynamique RECHERCHER, MINIMUM, MAXIMUM, SUCCESSEUR et PRÉDÉCESSEUR peuvent se faire en temps $O(h)$ sur un arbre binaire de recherche de hauteur h .*

Exercices

12.2.1 On suppose que des entiers compris entre 1 et 1000 sont disposés dans un arbre binaire de recherche et que l'on souhaite trouver le nombre 363. Parmi les séquences suivantes, lesquelles ne pourraient *pas* être la suite des nœuds parcourus ?

- 2, 252, 401, 398, 330, 344, 397, 363.
- 924, 220, 911, 244, 898, 258, 362, 363.
- 925, 202, 911, 240, 912, 245, 363.
- 2, 399, 387, 219, 266, 382, 381, 278, 363.
- 935, 278, 347, 621, 299, 392, 358, 363.

12.2.2 Écrire des versions récursives des procédures ARBRE-MINIMUM et ARBRE-MAXIMUM.

12.2.3 Écrire la procédure ARBRE-PRÉDÉCESSEUR.

12.2.4 Le professeur Szmoldu pense avoir découvert une remarquable propriété des arbres binaires de recherche. Supposez que la recherche d'une clé k dans un arbre binaire de recherche se termine sur une feuille. On considère trois ensembles : A , les clés situées à

gauche du chemin de recherche ; B , celles situées sur le chemin de recherche ; et C , les clés situées à droite du chemin de recherche. Le professeur Szmoldu affirme que, étant donnés trois $a \in A$, $b \in B$ et $c \in C$ quelconques, ils doivent satisfaire à $a \leq b \leq c$. Donner un contre-exemple qui soit le plus petit possible.

12.2.5 Montrer que, si un nœud d'un arbre binaire de recherche a deux enfants, alors son successeur n'a pas d'enfant de gauche et son prédécesseur n'a pas d'enfant de droite.

12.2.6 Soit un arbre binaire de recherche T dont les clés sont distinctes. Montrer que, si le sous-arbre de droite d'un nœud x dans T est vide et que x a un successeur y , alors y est l'ancêtre le plus bas de x dont l'enfant de gauche est aussi un ancêtre de x . (Ne pas oublier que chaque nœud est son propre ancêtre.)

12.2.7 On peut implémenter un parcours infixe d'un arbre à n nœuds en trouvant l'élément minimal de l'arbre à l'aide de ARBRE-MINIMUM, puis en appelant $n - 1$ fois ARBRE-SUCCESSEUR. Démontrer que cet algorithme s'exécute dans un temps $\Theta(n)$.

12.2.8 Démontrer que, quel que soit le nœud dont on part dans un arbre binaire de recherche de hauteur h , k appels successifs à ARBRE-SUCCESSEUR dépendent un temps $O(k + h)$.

12.2.9 Soit T un arbre binaire de recherche dont les clés sont distinctes, soit x un nœud feuille et soit y son parent. Montrer que $clé[y]$ est soit la plus petite clé de T supérieure à $clé[x]$, soit la plus grande clé de T inférieure à $clé[x]$.

12.3 INSERTION ET SUPPRESSION

Les opérations d'insertion et de suppression modifient l'ensemble dynamique représenté par un arbre binaire de recherche. La structure de données doit être modifiée pour refléter ce changement, tout en conservant la propriété d'arbre binaire de recherche. Comme nous le verrons, modifier l'arbre pour y insérer un nouvel élément est relativement simple, mais la suppression est un peu plus délicate à gérer.

a) Insertion

Pour insérer une nouvelle valeur v dans un arbre binaire de recherche T , on utilise la procédure ARBRE-INSÉRER. On lui passe un nœud z pour lequel $clé[z] = v$, $gauche[z] = \text{NIL}$ et $droite[z] = \text{NIL}$. Elle modifie T et certains champs de z pour permettre à z d'être inséré à sa position correcte dans l'arbre.

La figure 12.3 montre le fonctionnement de ARBRE-INSÉRER. Comme pour ARBRE-RECHERCHER et ARBRE-RECHERCHER-ITÉRATIF, ARBRE-INSÉRER part de la racine de l'arbre et suit un chemin descendant. Le pointeur x parcourt le chemin et le parent de x est conservé dans le pointeur y . Après l'initialisation, la boucle **tant que** des lignes 3–7 fait descendre ces deux pointeurs le long de l'arbre, bifurquant à gauche ou à droite en fonction du résultat de la comparaison entre $clé[z]$

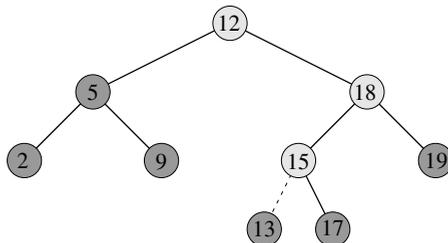


Figure 12.3 Insertion d'un élément de clé 13 dans un arbre binaire de recherche. Les nœuds sur fond gris clair indiquent le chemin descendant de la racine vers la position où l'élément sera inséré. La ligne pointillée représente le lien qui est ajouté à l'arbre pour insérer l'élément.

et $clé[x]$, jusqu'à ce que x prenne la valeur NIL. Ce NIL occupe la position où l'on souhaite placer l'élément d'entrée z . Les lignes 8–13 initialisent les pointeurs qui provoquent l'insertion de z .

ARBRE-INSÉRER(T, z)

```

1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{racine}[T]$ 
3  tant que  $x \neq \text{NIL}$ 
4      faire  $y \leftarrow x$ 
5          si  $clé[z] < clé[x]$ 
6              alors  $x \leftarrow \text{gauche}[x]$ 
7              sinon  $x \leftarrow \text{droite}[x]$ 
8   $p[z] \leftarrow y$ 
9  si  $y = \text{NIL}$ 
10     alors  $\text{racine}[T] \leftarrow z$             $\triangleright$  arbre  $T$  était vide
11     sinon si  $clé[z] < clé[y]$ 
12         alors  $\text{gauche}[y] \leftarrow z$ 
13         sinon  $\text{droite}[y] \leftarrow z$ 

```

Comme les autres opérations primitives d'arbre binaire de recherche, la procédure ARBRE-INSÉRER s'exécute dans un temps $O(h)$ sur un arbre de hauteur h .

b) Suppression

La procédure permettant de supprimer un nœud donné z d'un arbre binaire de recherche prend comme argument un pointeur sur z . La procédure considère les trois cas montrés à la figure 12.4. Si z n'a pas d'enfant, on modifie son parent $p[z]$ pour remplacer z par NIL dans le champ enfant. Si le nœud n'a qu'un seul enfant, on « détache » z en créant un nouveau lien entre son enfant et son parent. Enfin, si le nœud a deux enfants, on détache le successeur de z , y , qui n'a pas d'enfant gauche (voir exercice 12.2.5) et on remplace la clé et les données satellites de z par la clé et les données satellite de y .

Le code de ARBRE-SUPPRIMER gère ces trois cas un peu différemment.

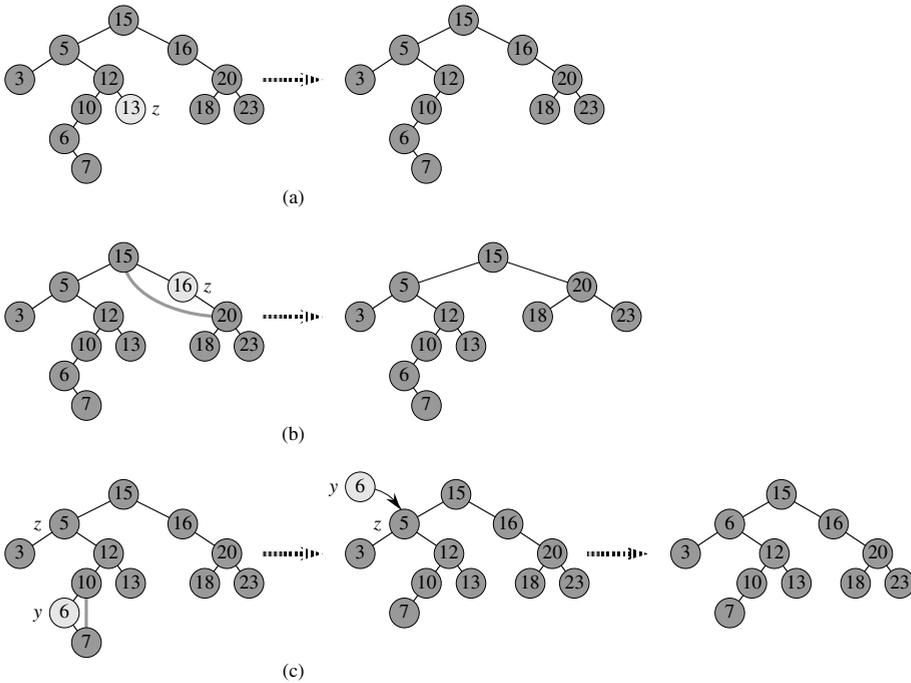


Figure 12.4 Suppression d'un nœud z d'un arbre binaire de recherche. Pour savoir quel est le nœud à supprimer effectivement, on regarde le nombre d'enfants de z ; ce nœud est colorié en gris clair. (a) Si z n'a pas d'enfant, on se contente de le supprimer. (b) Si z n'a qu'un seul enfant, on détache z . (c) Si z a deux enfants, on détache son successeur y qui possède au plus un enfant et on remplace alors la clé et les données satellites de z par celles de y .

ARBRE-SUPPRIMER(T, z)

- 1 **si** $gauche[z] = \text{NIL}$ ou $droite[z] = \text{NIL}$
- 2 **alors** $y \leftarrow z$
- 3 **sinon** $y \leftarrow \text{ARBRE-SUCCESSEUR}(z)$
- 4 **si** $gauche[y] \neq \text{NIL}$
- 5 **alors** $x \leftarrow gauche[y]$
- 6 **sinon** $x \leftarrow droite[y]$
- 7 **si** $x \neq \text{NIL}$
- 8 **alors** $p[x] \leftarrow p[y]$
- 9 **si** $p[y] = \text{NIL}$
- 10 **alors** $racine[T] \leftarrow x$
- 11 **sinon si** $y = gauche[p[y]]$
- 12 **alors** $gauche[p[y]] \leftarrow x$
- 13 **sinon** $droite[p[y]] \leftarrow x$
- 14 **si** $y \neq z$
- 15 **alors** $clé[z] \leftarrow clé[y]$
- 16 copier données satellites de y dans z
- 17 **retourner** y

Aux lignes 1–3, l’algorithme détermine un nœud y à détacher. Le nœud y est soit le nœud d’entrée z (si z a au plus 1 enfant), soit le successeur de z (si z a deux enfant). Puis, aux lignes 4–6, x est rendu égal à l’enfant non-NIL de y ou à NIL si y n’a pas d’enfant. Le nœud y est détaché aux lignes 7–13 via modification des pointeurs dans $p[y]$ et x . Détacher y est assez compliqué, du fait de la nécessité de bien gérer les conditions aux limites, qui surviennent lorsque $x = \text{NIL}$ ou quand y est la racine. Enfin, aux lignes 14–16, si le successeur de z était le nœud détaché, la clé et les données satellites de y sont déplacées dans z , écrasant alors la clé et les données satellites précédentes. Le nœud y est retourné en ligne 17 pour que la procédure appelante puisse le recycler *via* la liste libre. La procédure s’exécute en temps $O(h)$ sur un arbre de hauteur h .

En résumé, nous venons de démontrer le théorème suivant :

Théorème 12.3 *On peut exécuter les opérations INSÉRER et SUPPRIMER d’ensemble dynamique en temps $O(h)$ sur un arbre binaire de recherche de hauteur h .*

Exercices

12.3.1 Donner une version récursive de la procédure ARBRE-INSÉRER.

12.3.2 On suppose qu’un arbre binaire de recherche est construit par insertion répétée de valeurs distinctes dans l’arbre. Dire pourquoi le nombre de nœuds examinés lors de la recherche d’une valeur dans l’arbre vaut un de plus que le nombre de nœuds examinés quand la valeur a été insérée pour la première fois dans l’arbre.

12.3.3 On peut trier un ensemble donné de n nombres en commençant par construire un arbre binaire de recherche contenant ces nombres (en répétant ARBRE-INSÉRER pour insérer les nombres un à un), puis en imprimant les nombres *via* un parcours infixe de l’arbre. Quels sont les temps d’exécution de cet algorithme de tri, dans le pire et dans le meilleur des cas ?

12.3.4 On suppose qu’une autre structure de données contient un pointeur sur un nœud y d’un arbre binaire de recherche et que z , le prédécesseur de y est supprimé de l’arbre par la procédure ARBRE-SUPPRIMER. Quel problème peut-il se produire ? Comment peut-on réécrire ARBRE-SUPPRIMER pour résoudre ce problème ?

12.3.5 L’opération de suppression est-elle « commutative » au sens où la suppression de x puis de y dans un arbre binaire de recherche produit le même arbre que la suppression de y puis de x ? Si oui, dire pourquoi ; sinon, donner un contre-exemple.

12.3.6 Lorsque le nœud z dans ARBRE-SUPPRIMER a deux enfants, on pourrait détacher son prédécesseur plutôt que son successeur. Certains ont dit qu’une stratégie équilibrée, qui donnerait des priorités égales au prédécesseur et au successeur, donnerait des performances empiriques meilleures. Comment pourrait-on modifier ARBRE-SUPPRIMER pour implémenter cette stratégie équilibrée ?

12.4[★] ARBRES BINAIRES DE RECHERCHE CONSTRUITS ALÉATOIREMENT

Nous avons montré que toutes les opérations basiques d'arbre binaire de recherche se font en temps $O(h)$, h étant la hauteur de l'arbre. La hauteur d'un arbre binaire de recherche varie, cependant, au fur et à mesure que l'on ajoute ou supprime des éléments. Si, par exemple, on insère les éléments dans l'ordre strictement croissant, alors l'arbre sera une chaîne de hauteur $n - 1$. D'un autre côté, l'exercice B.5.4 montre que $h \geq \lceil \lg n \rceil$. Comme pour le tri rapide, on peut montrer que le comportement du cas moyen est beaucoup plus proche du cas optimal que du cas le plus défavorable.

Malheureusement, on sait peu de choses sur la hauteur moyenne d'un arbre binaire quand il est créé par des insertions et par des suppressions. Quand l'arbre n'est créé que par des insertions, l'analyse devient plus accessible. Définissons donc un *arbre binaire de recherche construit aléatoirement* à n clés comme étant un arbre qui résulte de l'ajout de clés dans un ordre aléatoire à un arbre initialement vide, chacune des $n!$ permutations des clés d'entrée étant équiprobable. (L'exercice 12.4.3 vous demandera de montrer que cette notion n'est pas la même chose que de supposer que chaque arbre binaire de recherche à n clés est équiprobable.) Cette section va démontrer que la hauteur attendue d'un arbre binaire de recherche à n clés construit aléatoirement est $O(\lg n)$. On supposera que toutes les clés sont distinctes.

Commençons par définir trois variables aléatoires qui facilitent la mesure de la hauteur d'un arbre binaire de recherche construit aléatoirement. On note X_n la hauteur d'un arbre binaire de recherche à n clés construit aléatoirement et l'on définit la *hauteur exponentielle* $Y_n = 2^{X_n}$. Quand on construit un arbre binaire de recherche à n clés, on choisit une clé comme clé de la racine et on note R_n la variable aléatoire qui contient le rang de cette clé dans l'ensemble des n clés. La valeur de R_n a une chance égale d'être l'un quelconque des éléments de l'ensemble $\{1, 2, \dots, n\}$. Si $R_n = i$, alors le sous-arbre de gauche de la racine est un arbre binaire de recherche à $i - 1$ clés construit aléatoirement et le sous-arbre de droite est un arbre binaire de recherche à $n - i$ clés construit aléatoirement. Comme la hauteur d'un arbre binaire est supérieure d'une unité à la plus grande des hauteurs des deux sous-arbres de la racine, la hauteur exponentielle d'un arbre binaire vaut deux fois le maximum des hauteurs exponentielles des deux sous-arbres de la racine. Si l'on sait que $R_n = i$, on a donc

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) .$$

Comme cas de base, on a $Y_1 = 1$ car la hauteur exponentielle d'un arbre à 1 nœud est $2^0 = 1$ et, par commodité, on définit $Y_0 = 0$.

On définit ensuite des variables indicatrices $Z_{n,1}, Z_{n,2}, \dots, Z_{n,n}$, où

$$Z_{n,i} = \mathbf{I}\{R_n = i\} .$$

Comme R_n a une chance égale d'être l'un quelconque des éléments de $\{1, 2, \dots, n\}$, on a $\Pr\{R_n = i\} = 1/n$ pour $i = 1, 2, \dots, n$; d'où, d'après le lemme 5.1,

$$E[Z_{n,i}] = 1/n, \quad (12.1)$$

pour $i = 1, 2, \dots, n$. Comme il y a exactement une seule valeur de $Z_{n,i}$ qui est 1 et que toutes les autres sont 0, on a également

$$Y_n = \sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i})) .$$

Nous allons montrer que $E[Y_n]$ est polynomial en n , ce qui impliquera au final que $E[X_n] = O(\lg n)$.

La variable indicatrice $Z_{n,i} = I\{R_n = i\}$ est indépendante des valeurs de Y_{i-1} et Y_{n-i} . En ayant choisi $R_n = i$, le sous-arbre de gauche, dont la hauteur exponentielle est Y_{i-1} , est construit aléatoirement sur les $i - 1$ clés dont les rangs sont inférieurs à i . Ce sous-arbre ressemble en tout point à n'importe quel autre arbre binaire de recherche construit aléatoirement sur $i - 1$ clés. À part le nombre de clés qu'elle contient, la structure de ce sous-arbre n'est nullement affectée par le choix de $R_n = i$; et donc, les variables aléatoires Y_{i-1} et $Z_{n,i}$ sont indépendantes. De même, le sous-arbre de droite, dont la hauteur exponentielle est Y_{n-i} , est construit aléatoirement sur les $n - i$ clés dont les rangs sont supérieurs à i . Sa structure est indépendante de la valeur de R_n , et donc les variables aléatoires Y_{n-i} et $Z_{n,i}$ sont indépendantes. D'où

$$\begin{aligned} E[Y_n] &= E\left[\sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\ &= \sum_{i=1}^n E[Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i}))] \quad (\text{d'après la linéarité de l'espérance}) \\ &= \sum_{i=1}^n E[Z_{n,i}] E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad (\text{d'après l'indépendance}) \\ &= \sum_{i=1}^n \frac{1}{n} \cdot E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad (\text{d'après l'équation (12.1)}) \\ &= \frac{2}{n} \sum_{i=1}^n E[\max(Y_{i-1}, Y_{n-i})] \quad (\text{d'après l'équation C.21}) \\ &\leq \frac{2}{n} \sum_{i=1}^n (E[Y_{i-1}] + E[Y_{n-i}]) \quad (\text{d'après l'exercice C.3.4}) . \end{aligned}$$

Chaque terme $E[Y_0], E[Y_1], \dots, E[Y_{n-1}]$ apparaît deux fois dans la dernière sommation, une fois en tant que $E[Y_{i-1}]$ et une fois en tant que $E[Y_{n-i}]$, d'où la récurrence

$$E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i]. \quad (12.2)$$

En employant la méthode par substitution, nous allons montrer que, pour tous les entiers positifs n , la récurrence (12.2) admet la solution

$$E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

Ce faisant, nous utiliserons l'identité

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}. \quad (12.3)$$

(L'exercice 12.4.1 vous demandera de prouver cette identité.)

Pour le cas de base, on vérifie que la borne

$$1 = Y_1 = E[Y_1] \leq \frac{1}{4} \binom{1+3}{3} = 1$$

est vérifiée. Pour la substitution, on a

$$\begin{aligned} E[Y_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \\ &\leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \quad (\text{d'après l'hypothèse de récurrence}) \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \\ &= \frac{1}{n} \binom{n+3}{4} \quad (\text{d'après l'équation (12.3)}) \\ &= \frac{1}{n} \cdot \frac{(n+3)!}{4! (n-1)!} \\ &= \frac{1}{4} \cdot \frac{(n+3)!}{3! n!} \\ &= \frac{1}{4} \binom{n+3}{3}. \end{aligned}$$

On a majoré $E[Y_n]$, mais l'objectif final est de borner $E[X_n]$. Ainsi que l'exercice 12.4.4 vous demandera de le montrer, la fonction $f(x) = 2^x$ est convexe (voir page 1072). On peut donc appliquer l'inégalité de Jensen (C.25) qui dit que

$$2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n] ,$$

pour en déduire que

$$\begin{aligned} 2^{E[X_n]} &\leq \frac{1}{4} \binom{n+3}{3} \\ &= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\ &= \frac{n^3 + 6n^2 + 11n + 6}{24} . \end{aligned}$$

En prenant les logarithmes des deux membres, on obtient $E[X_n] = O(\lg n)$. Ainsi, nous avons démontré le résultat suivant :

Théorème 12.4 *La hauteur moyenne d'un arbre binaire de recherche construit aléatoirement à partir de n clés est $O(\lg n)$.*

Exercices

12.4.1 Prouver l'équation (12.3).

12.4.2 Décrire un arbre binaire de recherche à n nœuds tel que la profondeur moyenne d'un nœud dans l'arbre soit $\Theta(\lg n)$, mais que la hauteur de l'arbre soit $\omega(\lg n)$. Donner un majorant asymptotique pour la hauteur d'un arbre binaire de recherche à n nœuds dans lequel la profondeur moyenne d'un nœud est $\Theta(\lg n)$.

12.4.3 Montrer que la notion d'arbre binaire de recherche choisi aléatoirement sur n clés, où chaque arbre binaire de recherche de n clés a une chance égale d'être choisi, diffère de la notion d'arbre binaire de recherche construit aléatoirement, vue dans cette section. (*conseil* : Énumérer les possibilités quand $n = 3$.)

12.4.4 Montrer que la fonction $f(x) = 2^x$ est convexe.

12.4.5 * On considère l'action de TRI-RAPIDE-RANDOMISE sur une séquence d'entrée de n nombres. Démontrer que, pour toute constante $k > 0$, toutes les $n!$ permutations de l'entrée, sauf $O(1/n^k)$ d'entre elles, engendrent un temps d'exécution $O(n \lg n)$.

PROBLÈMES

12.1. Arbres binaires de recherche contenant des clés égales

L'égalité entre clés pose un problème pour l'implémentation des arbres binaires de recherche.

- a. Quelles sont les performances asymptotiques de ARBRE-INSÉRER quand on l'utilise pour insérer n éléments ayant des clés identiques dans un arbre binaire de recherche initialement vide ?

On se propose d'améliorer ARBRE-INSÉRER en lui faisant tester avant la ligne 5 si $clé[z] = clé[x]$ et en lui faisant tester avant la ligne 11 si $clé[z] = clé[y]$. En cas d'égalité, on met en œuvre l'une des stratégies suivantes. Pour chaque stratégie, trouver les performances asymptotiques de l'insertion de n éléments ayant des clés identiques dans un arbre binaire de recherche initialement vide. (Les stratégies sont décrites pour la ligne 5, dans laquelle on compare les clés de z et x . Remplacer x par y pour adapter ces stratégies à la ligne 11.)

- b. Gérer un indicateur booléen $b[x]$ dans le nœud x . Initialiser x à *gauche*[x] ou à *droite*[x] selon la valeur de $b[x]$, qui alterne entre FAUX et VRAI chaque fois que le nœud est visité pendant l'insertion d'un nœud ayant la même clé que x .
- c. Gérer une liste de nœuds ayant des clés égales à x et insérer z dans la liste.
- d. Initialiser x aléatoirement à *gauche*[x] ou à *droite*[x]. (Donner les performances dans le cas le plus défavorable et en déduire, de manière informelle, les performances dans le cas moyen.)

12.2. Arbres à base

Étant données deux chaînes $a = a_0a_1 \dots a_p$ et $b = b_0b_1 \dots b_q$, où chaque a_i et chaque b_j appartiennent à un ensemble ordonné de caractères, on dit que la chaîne a est *lexicographiquement inférieure* à la chaîne b si

- 1) il existe un entier j , avec $0 \leq j \leq \min(p, q)$, tel que $a_i = b_i$ pour tout $i = 0, 1, \dots, j-1$ et $a_j < b_j$, ou
- 2) $p < q$ et $a_i = b_i$ pour tout $i = 0, 1, \dots, p$.

Par exemple, si a et b sont des chaînes de bits, alors $10100 < 10110$ d'après la règle 1 (en prenant $j = 3$) et $10100 < 101000$ d'après la règle 2. Cela équivaut à l'ordre en usage dans les dictionnaires.

La structure de données d'*arbre à base* montrée à la figure 12.5 contient les chaînes de bits 1011, 10, 011, 100 et 0. Lors d'une recherche d'une clé $a = a_0a_1 \dots a_p$, on se dirige vers la gauche au nœud de profondeur i si $a_i = 0$ et vers la droite si $a_i = 1$. Soit S un ensemble de chaînes binaires distinctes dont les longueurs ont pour cumul n . Montrer comment utiliser un arbre à base pour trier S lexicographiquement en temps

$\Theta(n)$. Pour l'exemple de la figure 12.5, la sortie du tri serait la séquence 0, 011, 10, 100, 1011.

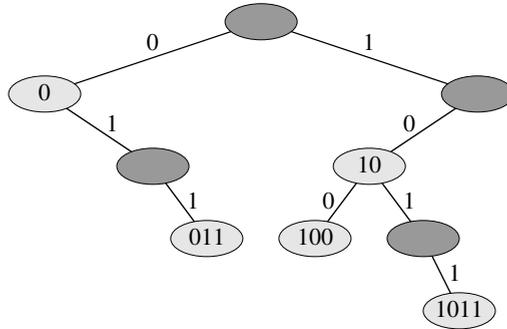


Figure 12.5 Un arbre à base contenant les chaînes de bits 1011, 10, 011, 100 et 0. La clé de chaque nœud peut être déterminée en empruntant le chemin reliant la racine à ce nœud. Il est donc inutile de conserver les clés dans les nœuds; les clés sont montrées ici uniquement pour des raisons de clarté. Les nœuds sont en gris foncé si les clés correspondantes ne sont pas dans l'arbre; ces nœuds servent uniquement à créer des liens entre d'autres nœuds.

12.3. Profondeur moyenne d'un nœud dans un arbre binaire de recherche construit aléatoirement

Dans ce problème, on démontre que la profondeur moyenne d'un nœud dans un arbre binaire de recherche construit aléatoirement à n nœuds est $O(\lg n)$. Bien que ce résultat soit moins fort que celui du théorème 12.4, la technique que nous allons utiliser révèle une surprenante ressemblance entre la construction d'un arbre binaire de recherche et l'exécution de TRI-RAPIDE-RANDOMISÉ vu à la section 7.3.

On définit la **longueur de chemin totale** $P(T)$ d'un arbre binaire T comme étant la somme, prise sur tous les nœuds x de T , de la profondeur du nœud x notée $d(x, T)$.

a. Montrer que la profondeur moyenne d'un nœud dans T est

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Ainsi, on veut montrer que la valeur attendue de $P(T)$ est $O(n \lg n)$.

b. Soient T_G et T_D les sous-arbres de gauche et de droite de l'arbre T . Montrer que, si T possède n nœuds, alors

$$P(T) = P(T_G) + P(T_D) + n - 1.$$

c. Soit $P(n)$ la longueur de chemin totale moyenne d'un arbre binaire de recherche à n nœuds construit aléatoirement. Montrer que

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n - i - 1) + n - 1).$$

d. Montrer que $P(n)$ peut être réécrit sous la forme

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n).$$

e. En s'inspirant de l'analyse alternative de la version randomisée du tri rapide (voir problème 7.2, conclure que $P(n) = O(n \lg n)$.

A chaque appel récursif du tri rapide, on choisit au hasard un élément pivot pour partitionner l'ensemble des éléments en cours de tri. Chaque nœud d'un arbre binaire de recherche crée une partition à partir de l'ensemble des éléments qui se trouvent dans le sous-arbre enraciné à ce nœud.

f. Décrire une implémentation du tri rapide dans laquelle les comparaisons servant à trier un ensemble d'éléments sont exactement les mêmes que celles qui servent à insérer les éléments dans un arbre binaire de recherche. (L'ordre dans lequel sont effectuées les comparaisons peut changer, mais les comparaisons elles-mêmes doivent être identiques).

12.4. Nombre d'arbres binaires différents

Soit b_n le nombre d'arbres binaires différents à n nœuds. Dans ce problème, on devra trouver une formule donnant b_n ainsi qu'une estimation asymptotique.

a. Montrer que $b_0 = 1$ et que, pour $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

b. Soit $B(x)$ la fonction génératrice

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

(voir problème 4.5 pour la définition d'une fonction génératrice). Montrer que $B(x) = xB(x)^2 + 1$, et donc qu'une façon d'exprimer $B(x)$ sous forme fermée est

$$B(x) = \frac{1}{2x} (1 - \sqrt{1 - 4x}).$$

Le *développement de Taylor* de $f(x)$ autour du point $x = a$ est donné par

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x - a)^k,$$

où $f^{(k)}(x)$ est la dérivée k ème de f en x .

c. Montrer que

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

(*nième nombre de Catalan*) en utilisant le développement de Taylor de $\sqrt{1-4x}$ autour de $x = 0$. (A la place du développement de Taylor, on peut utiliser la généralisation du développement binomial (C.4) aux exposants n non entiers : pour un nombre réel n et un entier k quelconques, on interprète $\binom{n}{k}$ comme étant $n(n-1)\cdots(n-k+1)/k!$ si $k \geq 0$ et 0 sinon.)

d. Montrer que

$$b_n = \frac{4^n}{\sqrt{\pi n^{3/2}}} (1 + O(1/n)) .$$

NOTES

Knuth [185] contient une bonne étude des arbres binaires de recherche simples et de nombreuses variantes. Les arbres binaires de recherche semblent avoir été découverts indépendamment par plusieurs personnes vers la fin des années 1950. Knuth [185] traite aussi des arbres à base.

La section 15.5 montrera comment construire un arbre binaire de recherche optimal quand les fréquences de recherche sont connues avant la construction de l'arbre. En d'autres termes : connaissant les fréquences de recherche pour chaque clé et les fréquences de recherche pour les valeurs qui tombent entre des clés de l'arbre, on construit un arbre binaire de recherche tel qu'un ensemble de recherches qui suit ces fréquences examinera le nombre minimal de nœuds.

La démonstration donnée à la section 12.4 qui borne la hauteur moyenne d'un arbre binaire de recherche construit aléatoirement est due à Aslam [23]. Martínez et Roura [211] donnent des algorithmes randomisés pour l'insertion et la suppression dans les arbres binaires de recherche dans lesquels le résultat de l'une ou l'autre de ces deux opérations est un arbre binaire de recherche aléatoire. Leur définition d'un arbre binaire de recherche aléatoire diffère légèrement, cependant, de celle d'un arbre binaire de recherche construit aléatoirement telle que donnée dans ce chapitre.

Chapitre 13

Arbres rouge-noir

Le chapitre 12 a montré qu'un arbre binaire de recherche de hauteur h permettait de mettre en œuvre les opérations fondamentales d'ensemble dynamique, telles RECHERCHER, PRÉDÉCESSEUR, SUCCESSEUR, MINIMUM, MAXIMUM, INSÉRER et SUPPRIMER, avec un temps $O(h)$. Les opérations sont donc d'autant plus rapides que la hauteur de l'arbre est petite ; mais si cette hauteur est grande, les performances risquent de ne pas être meilleures qu'avec une liste chaînée. Les arbres rouge-noir sont l'un des nombreux modèles d'arbres de recherche « équilibrés », qui permettent aux opérations d'ensemble dynamique de s'exécuter en temps $O(\lg n)$ dans le cas le plus défavorable.

13.1 PROPRIÉTÉS DES ARBRES ROUGE-NOIR

Un *arbre rouge-noir* est un arbre binaire de recherche comportant un bit de stockage supplémentaire par nœud : sa *couleur*, qui peut être ROUGE ou NOIR. En contrôlant la manière dont les nœuds sont coloriés sur n'importe quel chemin allant de la racine à une feuille, les arbres rouge-noir garantissent qu'aucun de ces chemins n'est plus de deux fois plus long que n'importe quel autre, ce qui rend l'arbre approximativement *équilibré*.

Chaque nœud de l'arbre contient maintenant les champs *couleur*, *clé*, *gauche*, *droite* et *p*. Si un enfant ou le parent d'un nœud n'existe pas, le champ pointeur correspondant du nœud contient la valeur NIL. Nous considérerons ces NIL comme des pointeurs vers des nœuds externes (feuilles) de l'arbre binaire de recherche et les nœuds normaux, dotés de clés, comme des nœuds internes de l'arbre.

Un arbre binaire de recherche est un arbre rouge-noir s'il satisfait aux propriétés suivantes :

- 1) Chaque nœud est soit rouge, soit noir.
- 2) La racine est noire.
- 3) Chaque feuille (NIL) est noire.
- 4) Si un nœud est rouge, alors ses deux enfants sont noirs.
- 5) Pour chaque nœud, tous les chemins reliant le nœud à des feuilles contiennent le même nombre de nœuds noirs.

On peut voir un exemple d'arbre rouge-noir à la figure 13.1.

Pour simplifier le traitement des conditions aux limites dans la programmation des arbres rouge-noir, on utilise une même sentinelle pour représenter NIL (voir page 201). Pour un arbre rouge-noir T , la sentinelle $nil[T]$ est un objet ayant les mêmes champs qu'un nœud ordinaire. Son champ *couleur* vaut NOIR, et ses autres champs (*p*, *gauche*, *droite* et *clé*) peuvent prendre des valeurs quelconques. Comme le montre la figure 13.1(b), tous les pointeurs vers NIL sont remplacés par des pointeurs vers la sentinelle $nil[T]$.

On utilise la sentinelle pour pouvoir traiter un enfant NIL d'un nœud x comme un nœud ordinaire dont le parent est x . On pourrait ajouter un nœud sentinelle distinct pour chaque NIL de l'arbre, pour que le parent de chaque NIL soit bien défini, mais cette méthode gaspillerait de l'espace. À la place, on emploie une seule sentinelle $nil[T]$ pour représenter tous les NIL (à savoir les feuilles et le parent de la racine). Les valeurs des champs *p*, *gauche*, *droite* et *clé* de la sentinelle sont immatérielles, bien que nous puissions les configurer à notre convenance dans le cours d'une procédure.

On ne s'intéresse généralement qu'aux nœuds internes d'un arbre rouge-noir, vu que ce sont eux qui contiennent les valeurs de clé. Dans le reste du chapitre, nous omettrons les feuilles quand nous dessinerons des arbres rouge-noir, comme c'est le cas sur la figure 13.1(c).

On appelle *hauteur noire* le nombre de nœuds noirs d'un chemin partant d'un nœud x (non compris ce nœud) vers une feuille, et on utilise la notation $bh(x)$. D'après la propriété 5, la notion de hauteur noire est bien définie, puisque tous les chemins descendant d'un nœud contiennent le même nombre de nœuds noirs. On définit la hauteur noire d'un arbre rouge-noir comme étant la hauteur noire de sa racine. Le lemme suivant montre pourquoi les arbres rouge-noir sont de bons arbres de recherche.

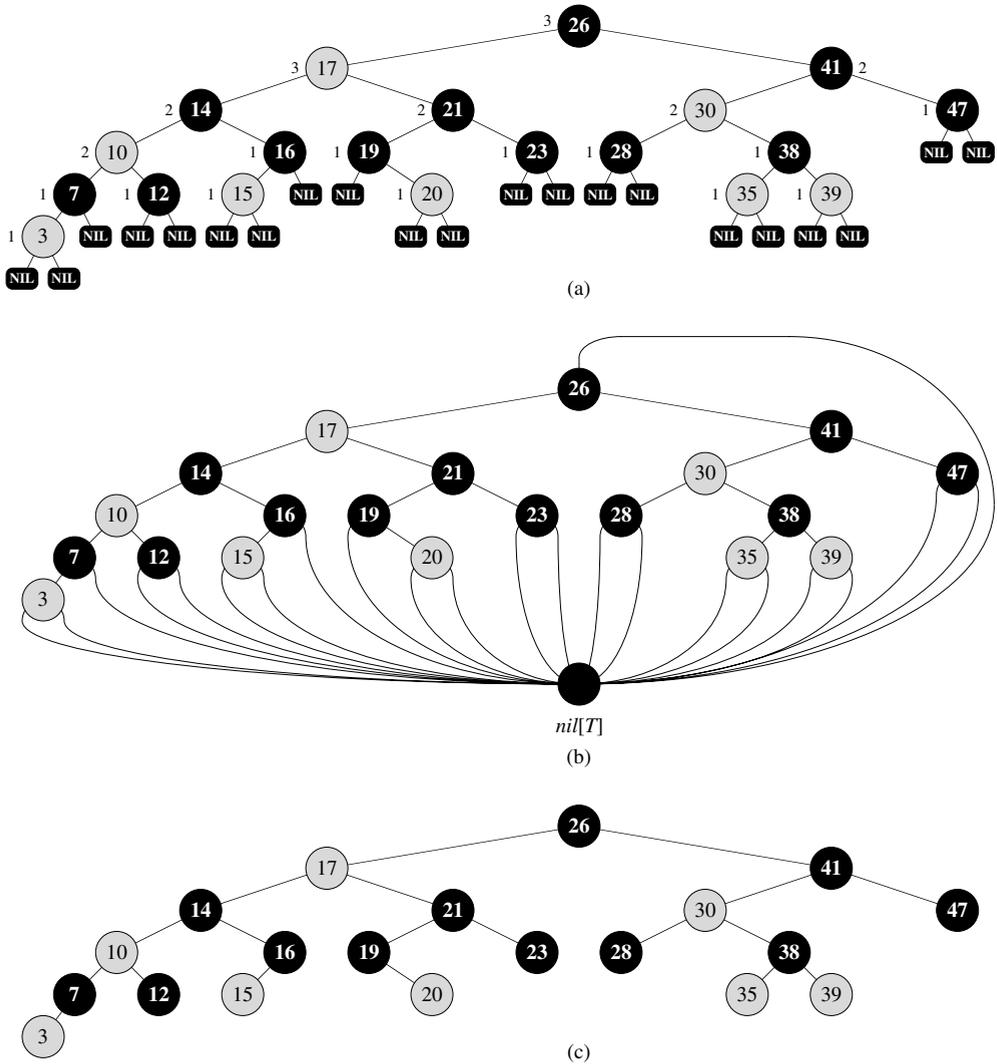


Figure 13.1 Un arbre rouge-noir dont les nœuds noirs sont représentés en noir, et les nœuds rouges en gris. Chaque nœud d'un arbre rouge-noir est rouge ou noir, les enfants d'un nœud rouge sont noirs, et chaque chemin simple reliant un nœud à une feuille contient le même nombre de nœuds noirs. (a) Chaque feuille, représentée par (NIL), est noire. Chaque nœud non-NIL est étiqueté par sa hauteur noire ; les NIL ont une hauteur noire égale à 0. (b) Le même arbre rouge-noir, mais où chaque NIL est remplacé par la sentinelle $nil[T]$, qui est toujours noire, et où les hauteurs noires sont omises. Le parent de la racine est aussi la sentinelle. (c) Le même arbre rouge-noir, mais où les feuilles et le parent de la racine ont été entièrement omis. Nous emploierons ce style de représentation dans le reste du chapitre.

Lemme 13.1 *Un arbre rouge-noir ayant n nœuds internes a une hauteur au plus égale à $2 \lg(n + 1)$.*

Démonstration : Commençons par montrer que le sous-arbre enraciné en un nœud x quelconque contient au moins $2^{\text{bh}(x)} - 1$ nœuds internes. Cette affirmation peut se démontrer par récurrence sur la hauteur de x . Si la hauteur de x est 0, alors x est obligatoirement une feuille ($\text{nil}[T]$) et le sous-arbre enraciné en x contient effectivement au moins $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 0$ nœuds internes. Pour l'étape inductive, soit un nœud interne x de hauteur positive ayant deux enfants. Chaque enfant a une hauteur noire $\text{bh}(x)$ ou $\text{bh}(x) - 1$, selon que sa couleur est rouge ou noire. Comme la hauteur d'un enfant de x est inférieure à celle de x lui-même, on peut appliquer l'hypothèse de récurrence pour conclure que chaque enfant a au moins $2^{\text{bh}(x)-1} - 1$ nœuds internes. Le sous-arbre de racine x contient donc au moins $(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ nœuds internes, ce qui démontre l'assertion.

Pour compléter la preuve du lemme, appelons h la hauteur de l'arbre. D'après la propriété 4, au moins la moitié des nœuds d'un chemin simple reliant la racine à une feuille, racine non comprise, doivent être noirs. En conséquence, la hauteur noire de la racine doit valoir au moins $h/2$; donc,

$$n \geq 2^{h/2} - 1 .$$

En faisant passer le 1 dans le membre gauche et en prenant le logarithme des deux membres, on obtient $\lg(n + 1) \geq h/2$, soit $h \leq 2 \lg(n + 1)$. \square

Une conséquence immédiate de ce lemme est que les opérations d'ensemble dynamique RECHERCHER, MINIMUM, MAXIMUM, SUCCESSEUR et PRÉDÉCESSEUR peuvent être implémentées en temps $O(\lg n)$ sur les arbres rouge-noir, puisqu'elles peuvent s'exécuter en temps $O(h)$ sur un arbre de recherche de hauteur h (comme on l'a vu au chapitre 12) et qu'un arbre rouge-noir à n nœuds est un arbre de recherche de hauteur $O(\lg n)$. (Bien évidemment, les références à NIL dans les algorithmes du chapitre 12 devraient être remplacées par $\text{nil}[T]$.) Bien que les algorithmes ARBRE-INSÉRER et ARBRE-SUPPRIMER du chapitre 12 s'exécutent en temps $O(\lg n)$ quand on leur fournit un arbre rouge-noir en entrée, les opérations INSÉRER et SUPPRIMER d'ensemble dynamique ne garantissent pas que l'arbre binaire de recherche modifié par elles restera un arbre rouge-noir. Nous verrons cependant aux sections 13.3 et 13.4 que ces deux opérations peuvent être effectivement gérées avec un temps $O(\lg n)$.

Exercices

13.1.1 En prenant modèle sur la figure 13.1(a), dessiner l'arbre binaire de recherche complet de hauteur 3 contenant les clés $\{1, 2, \dots, 15\}$. Ajouter les feuilles NIL et colorier les nœuds de trois manières différentes, de telle façon que les hauteurs noires des arbres rouge-noir résultants soient 2, 3 et 4.

13.1.2 Dessiner l'arbre rouge-noir qui résulte de l'appel à ARBRE-INSÉRER sur l'arbre de la figure 13.1 avec la valeur de clé 36. Si le nœud inséré est colorié en rouge, est-ce que l'arbre résultant est encore un arbre rouge-noir ? Et si le nœud est colorié en noir ?

13.1.3 Un *arbre rouge-noir relâché* est un arbre binaire de recherche qui satisfait aux propriétés rouge-noir 1, 3, 4 et 5. Autrement dit, la racine peut être rouge ou noire. Soit un arbre rouge-noir relâché T dont la racine est rouge. Si on colorie la racine de T en noir sans faire d'autre changement, est-ce que l'arbre résultant est un arbre rouge-noir ?

13.1.4 Supposons que l'on « absorbe » chaque nœud rouge d'un arbre rouge-noir dans son parent noir, de façon que les enfants du nœud rouge deviennent des enfants du parent noir. (On ne se préoccupe pas de ce qu'il advient des clés.) Quels sont les degrés possibles d'un nœud noir après absorption de tous ses enfants rouges ? Que peut-on dire des profondeurs des feuilles de l'arbre résultant ?

13.1.5 Montrer que le chemin simple le plus long reliant un nœud x d'un arbre rouge-noir à une feuille a une longueur qui est au plus égale à deux fois celle du plus court chemin simple reliant le nœud x à une feuille.

13.1.6 Quel est le plus grand nombre possible de nœuds internes d'un arbre rouge-noir de hauteur noire k ? Quel est le plus petit nombre possible ?

13.1.7 Décrire un arbre rouge-noir à n clés qui possède le ratio maximal entre nœud internes rouges et nœud internes noirs. Quel est ce ratio ? Quel est l'arbre qui a le ratio minimal, et quel est ce ratio ?

13.2 ROTATION

Les opérations d'arbre de recherche ARBRE-INSÉRER et ARBRE-SUPPRIMER, quand on les exécute sur un arbre rouge-noir à n clés, prennent un temps $O(\lg n)$. Comme elles modifient l'arbre, le résultat pourrait violer les propriétés d'arbre rouge-noir énumérées à la section 13.1. Pour restaurer ces propriétés, il faut changer les couleurs de certains nœuds de l'arbre et également modifier la chaîne des pointeurs.

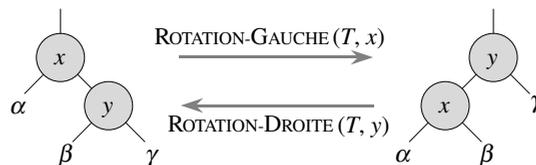


Figure 13.2 Les opérations de rotation sur un arbre de recherche binaire. L'opération $\text{ROTATION-GAUCHE}(T, x)$ transforme la configuration des deux nœuds de gauche pour aboutir à celle de droite, en modifiant un nombre constant de pointeurs. La configuration de droite peut être transformée en celle de gauche par l'opération inverse $\text{ROTATION-DROITE}(T, y)$. Les deux nœuds peuvent se trouver n'importe où dans l'arbre binaire de recherche. Les lettres α , β et γ représentent des sous-arbres arbitraires. Une opération de rotation préserve la propriété d'arbre binaire de recherche : les clés de α précèdent $\text{clé}[x]$, qui précède les clés de β , qui précède $\text{clé}[y]$, qui précède les clés de γ .

On modifie la chaîne des pointeurs *via* une **rotation**, qui est une opération locale d'arbre de recherche qui préserve la propriété d'arbre binaire de recherche. La figure 13.2 montre les deux sortes de rotation : les rotations gauches et les rotations droites. Lorsqu'on effectue une rotation gauche sur un nœud x , on suppose que son enfant de droite y n'est pas $nil[T]$; x peut être un nœud quelconque de l'arbre dont l'enfant de droite n'est pas $nil[T]$. La rotation gauche fait « pivoter » autour du lien qui relie x et y . Elle fait ensuite de y la nouvelle racine du sous-arbre, avec x qui devient l'enfant de gauche de y et l'enfant de gauche de y qui devient l'enfant de droite de x .

Le pseudo code de ROTATION-GAUCHE suppose que $droite[x] \neq nil[T]$ et que le parent de la racine est $nil[T]$.

```

ROTATION-GAUCHE( $T, x$ )
1   $y \leftarrow droite[x]$            ▷ initialise  $y$ .
2   $droite[x] \leftarrow gauche[y]$    ▷ sous-arbre gauche de  $y$  devient
                                   sous-arbre droit de  $x$ .

3  si  $gauche[y] \neq nil[T]$ 
4    alors  $p[gauche[y]] \leftarrow x$ 
5           $p[y] \leftarrow p[x]$      ▷ relie parent de  $x$  à  $y$ .
6          si  $p[x] = nil[T]$ 
7            alors  $racine[T] \leftarrow y$ 
8            sinon si  $x = gauche[p[x]]$ 
9              alors  $gauche[p[x]] \leftarrow y$ 
10             sinon  $droite[p[x]] \leftarrow y$ 
11           $gauche[y] \leftarrow x$    ▷ place  $x$  à gauche de  $y$ .
12           $p[x] \leftarrow y$ 

```

La figure 13.3 montre l'action de ROTATION-GAUCHE. Le code de ROTATION-DROITE est symétrique. Les deux procédures s'exécutent en temps $O(1)$. Les pointeurs sont les seuls champs du nœud qui sont modifiés par une rotation.

Exercices

13.2.1 Écrire le pseudo code de ROTATION-DROITE.

13.2.2 Montrer que, dans tout arbre binaire de recherche à n nœuds, il existe exactement $n - 1$ rotations possibles.

13.2.3 Soient a , b et c des nœuds arbitraires des sous-arbres α , β et γ (respectivement) de l'arbre de gauche de la figure 13.2. De quelle manière les profondeurs de a , b et c sont-elles modifiées lorsqu'on effectue une rotation gauche sur le nœud x de la figure ?

13.2.4 Montrer que tout arbre binaire de recherche à n nœuds peut être transformé en n'importe quel autre arbre binaire de recherche à n nœuds, à l'aide de $O(n)$ rotations. (*Conseil* : Commencer par montrer qu'au plus $n - 1$ rotations droite suffisent à transformer l'arbre en une chaîne orientée vers la droite.)

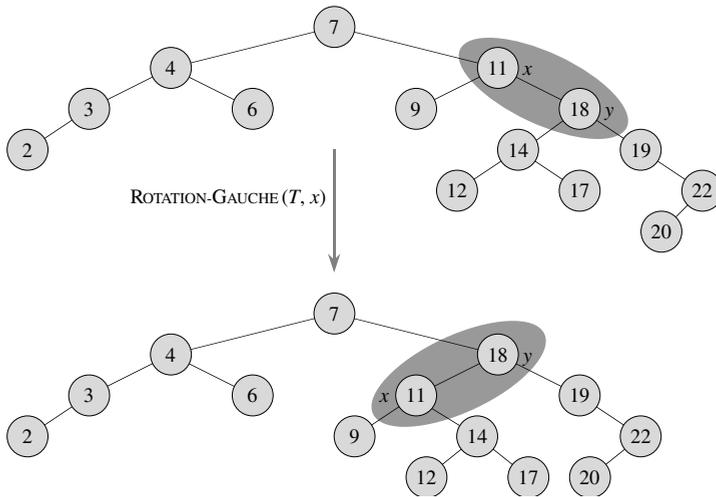


Figure 13.3 Un exemple de modification d'arbre binaire de recherche par la procédure $\text{ROTATION-GAUCHE}(T, x)$. La liste des valeurs de clé produite par un parcours infixe est la même pour l'arbre d'entrée que pour l'arbre modifié.

13.2.5 ★ On dit qu'un arbre binaire de recherche T_1 peut être *converti à droite* pour donner un arbre binaire de recherche T_2 s'il est possible d'obtenir T_2 à partir de T_1 via une série d'appels à ROTATION-DROITE . Donner un exemple de deux arbres T_1 et T_2 tels que T_1 ne peut pas être converti à droite pour donner T_2 . Montrer ensuite que, si un arbre T_1 peut être converti à droite pour donner T_2 , il peut être converti à droite à l'aide de $O(n^2)$ appels à ROTATION-DROITE .

13.3 INSERTION

L'insertion d'un nœud dans un arbre rouge-noir à n nœuds peut se faire en temps $O(\lg n)$. Nous utiliserons une version légèrement modifiée de la procédure ARBRE-INSÉRER (section 12.3) pour insérer le nœud z dans l'arbre T comme si c'était un arbre binaire ordinaire, puis nous colorierons z en rouge. Pour garantir la préservation des propriétés rouge-noir, nous appellerons ensuite la procédure auxiliaire $\text{RN-INSÉRER-CORRECTION}$ pour recolorier des nœuds et faire des rotations. L'appel $\text{RN-INSÉRER}(T, z)$ insère le nœud z , dont le champ *clé* est censé avoir été déjà rempli, dans l'arbre rouge-noir T .

$\text{RN-INSÉRER}(T, z)$

```

1   $y \leftarrow \text{nil}[T]$ 
2   $x \leftarrow \text{racine}[T]$ 
3  tant que  $x \neq \text{nil}[T]$ 
4      faire  $y \leftarrow x$ 
5          si  $\text{clé}[z] < \text{clé}[x]$ 
6              alors  $x \leftarrow \text{gauche}[x]$ 
7              sinon  $x \leftarrow \text{droite}[x]$ 

```

```

8  p[z] ← y
9  si y = nil[T]
10  alors racine[T] ← z
11  sinon si clé[z] < clé[y]
12      alors gauche[y] ← z
13      sinon droite[y] ← z
14  gauche[z] ← nil[T]
15  droite[z] ← nil[T]
16  couleur[z] ← ROUGE
17  RN-INSÉRER-CORRECTION(T, z)

```

Il y a quatre différences entre les procédures ARBRE-INSÉRER et RN-INSÉRER. Primo, toutes les instances de NIL dans ARBRE-INSÉRER sont remplacés par *nil*[T]. Secundo, on initialise *gauche*[z] et *droite*[z] à *nil*[T] aux lignes 14–15 de RN-INSÉRER, de façon à conserver la bonne structure d’arbre. Tertio, on colorie z en rouge à la ligne 16. Quarto, comme le coloriage de z en rouge risque de créer une violation d’une des propriétés rouge-noir, on appelle RN-INSÉRER-CORRECTION(T, z) en ligne 17 de RN-INSÉRER pour restaurer les propriétés rouge-noir.

RN-INSÉRER-CORRECTION(T, z)

```

1  tant que couleur[p[z]] = ROUGE
2      faire si p[z] = gauche[p[p[z]]]
3          alors y ← droite[p[p[z]]]
4              si couleur[y] = ROUGE
5                  alors couleur[p[z]] ← NOIR                ▷ Cas 1
6                  couleur[y] ← NOIR                          ▷ Cas 1
7                  couleur[p[p[z]]] ← ROUGE                  ▷ Cas 1
8                  z ← p[p[z]]                                ▷ Cas 1
9              sinon si z = droite[p[z]]
10                 alors z ← p[z]                              ▷ Cas 2
11                     ROTATION-GAUCHE(T, z)                 ▷ Cas 2
12                     couleur[p[z]] ← NOIR                  ▷ Cas 3
13                     couleur[p[p[z]]] ← ROUGE              ▷ Cas 3
14                     ROTATION-DROITE(T, p[p[z]])           ▷ Cas 3
15                 sinon (idem clause alors avec
                        permutation de « droite » et « gauche »)
16  couleur[racine[T]] ← NOIR

```

Pour comprendre le fonctionnement de RN-INSÉRER-CORRECTION, nous allons diviser notre étude du code en trois grandes phases. Primo, nous allons voir quelles sont les violations de propriété rouge-noir qui se produisent dans RN-INSÉRER quand le nœud z est inséré et colorié en rouge. Secundo, nous allons examiner le but général de la boucle **tant que** des lignes 1–15. Enfin, nous étudierons chacun des trois cas ⁽¹⁾

(1) Le cas 2 se ramène au cas 3, et ces deux cas ne sont donc pas mutuellement exclusifs.

constituant le corps de la boucle **tant que** et verrons comment ils fonctionnent. La figure 13.4 montre le fonctionnement de RN-INSÉRER-CORRECTION sur un exemple d'arbre rouge-noir.

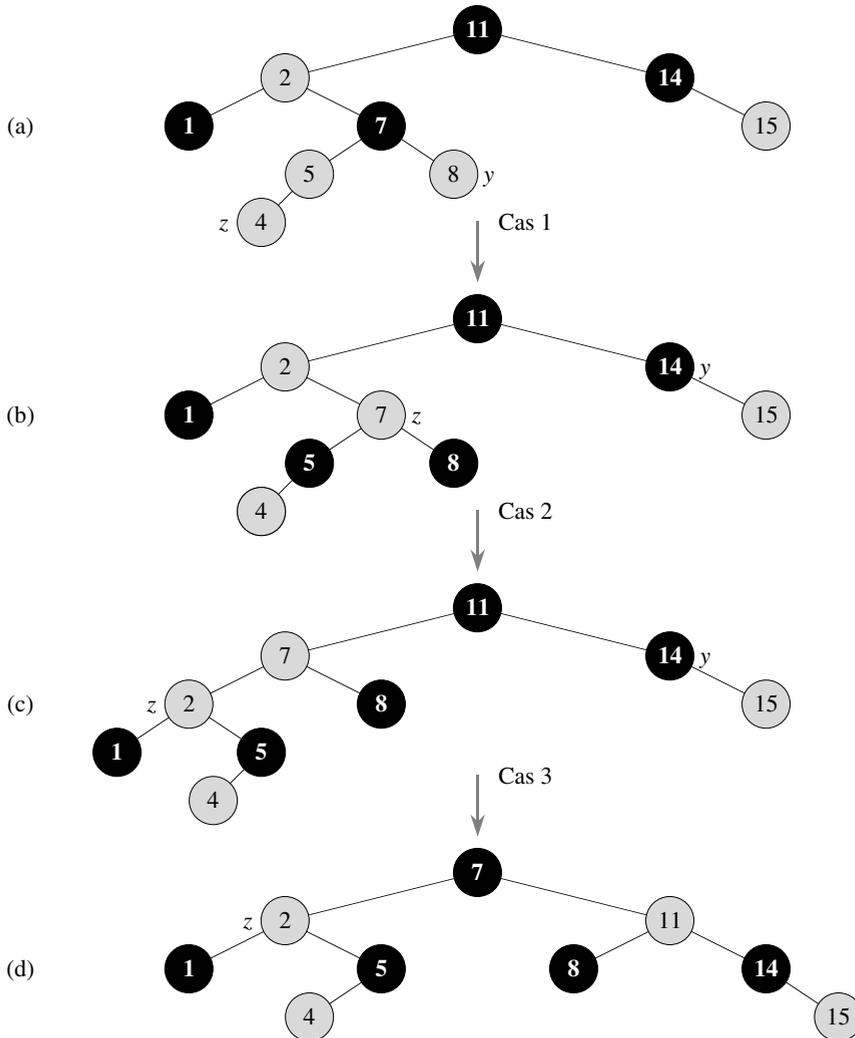


Figure 13.4 Fonctionnement de RN-INSÉRER-CORRECTION. (a) Un nœud z après insertion. Comme z et son parent $p[z]$ sont tous les deux rouges, il y a violation de la propriété 4. Comme l'oncle de z , y , est rouge, on est dans le cas 1 du code. Les nœuds sont recoloriés et le pointeur z remonte dans l'arbre, ce qui donne l'arbre montré en (b). Ici aussi, z et son parent sont tous les deux rouges, mais l'oncle de z , y , est noir. Comme z est l'enfant de droite de $p[z]$, on est dans le cas 2. On effectue une rotation gauche, et l'arbre résultant est montré en (c). Maintenant z est l'enfant de gauche de son parent, et l'on est dans le cas 3. Une rotation droite produit l'arbre de (d), qui est un arbre rouge-noir correct.

Quelle est celle des propriétés rouge-noir qui risque d'être violée lors de l'appel à RN-INSÉRER-CORRECTION ? La propriété 1 reste certainement vraie, de même que la propriété 3, car les deux enfants du nœud rouge nouvellement inséré sont la sentinelle $nil[T]$. La propriété 5, qui dit que le nombre de nœuds noirs est le même sur tous les chemins partant d'un nœud donné, est vérifiée elle aussi, car le nœud z remplace la sentinelle (noire) et le nœud z est rouge avec des enfants sentinelle. Donc, les seules propriétés susceptibles d'être violées sont la propriété 2, qui impose à la racine d'être noire, et la propriété 4, qui dit qu'un nœud rouge ne peut pas avoir d'enfant rouge. Ces deux infractions possibles sont dues au coloriage de z en rouge. La propriété 2 est violée si z est la racine, et la propriété 4 est violée si le parent de z est rouge. La figure 13.4(a) montre une infraction à la propriété 4 après que le nœud z a été inséré.

La boucle **tant que** des lignes 1–15 conserve l'invariant tripartite suivant :

Au début de chaque itération de la boucle,

- a) Le nœud z est rouge.
- b) Si $p[z]$ est la racine, alors $p[z]$ est noir.
- c) S'il y a violation des propriétés rouge-noir, il y a une violation au plus, et c'est une violation de la propriété 2 ou de la propriété 4. S'il y a violation de la propriété 2, cela vient de ce que z est la racine et est rouge. S'il y a violation de la propriété 4, cela vient de ce que z et $p[z]$ sont tous les deux rouges.

La partie (c), qui traite des violations des propriétés rouge-noir, est plus cruciale, pour montrer que RN-INSÉRER-CORRECTION restaure les propriétés rouge-noir, que les parties (a) et (b), qui nous serviront à l'occasion à comprendre des situations du code. Comme nous allons nous concentrer sur le nœud z et les nœuds proches de celui-ci dans l'arbre, il sera utile de savoir que, d'après la partie (a), z est rouge. Nous utiliserons la partie (b) pour montrer que le nœud $p[p[z]]$ existe quand on le référence aux lignes 2, 3, 7, 8, 13 et 14.

N'oubliez pas qu'il faut montrer qu'un invariant de boucle est vrai avant la première itération de la boucle, que chaque itération conserve l'invariant, et que l'invariant fournit une propriété utile à la fin de l'exécution de la boucle.

Commençons par l'initialisation et la fin de l'exécution. Ensuite, quand nous étudierons en détail le fonctionnement du corps de la boucle, nous démontrerons que la boucle conserve l'invariant à chaque itération. Au passage, nous démontrerons également qu'il y a deux possibilités à chaque itération de la boucle : le pointeur z remonte dans l'arbre, ou bien il y a des rotations qui se font et la boucle se termine.

Initialisation : Avant la première itération de la boucle, on avait un arbre rouge-noir sans violations auquel on avait ajouté un nœud rouge z . On va montrer que

chaque partie de l'invariant est vraie au moment où est appelée RN-INSÉRER-CORRECTION :

- a) Quand RN-INSÉRER-CORRECTION est appelée, z est le nœud rouge qui avait été ajouté.
- b) Si $p[z]$ est la racine, alors $p[z]$ avait commencé noir et n'avait pas changé avant l'appel de RN-INSÉRER-CORRECTION.
- c) Nous avons déjà vu que les propriétés 1, 3 et 5 sont vérifiées quand RN-INSÉRER-CORRECTION est appelée.

S'il y a violation de la propriété 2, alors la racine rouge est forcément le nœud nouvellement ajouté z , qui est le seul nœud interne de l'arbre. Comme le parent et les deux enfants de z sont la sentinelle, qui est noire, il n'y a pas violation de la propriété 4. Donc, la violation de la propriété 2 est la seule violation des propriétés rouge-noir dans tout l'arbre.

S'il y a violation de la propriété 4, alors, comme les enfants du nœud z sont des sentinelles noires et que l'arbre était correct avant l'ajout de z , la violation provient forcément de ce que z et $p[z]$ sont rouges. En outre, il n'y a pas d'autres violations des propriétés rouge-noir.

Terminaison : Quand la boucle se termine, elle le fait parce que $p[z]$ est noir. (Si z est la racine, alors $p[z]$ est la sentinelle $nil[T]$ qui est noire.) Donc, il n'y a pas violation de la propriété 4 après exécution de la boucle. D'après l'invariant de boucle, la seule propriété qui pourrait ne pas être vérifiée est la propriété 2. La ligne 16 restaure cette propriété, elle aussi, de sorte que, quand RN-INSÉRER-CORRECTION prend fin, toutes les propriétés rouge-noir sont vérifiées.

Conservation : Il y a en fait six cas à considérer dans la boucle **tant que**, mais trois d'entre eux sont symétriques par rapport aux trois autres, selon que le parent de z , $p[z]$, est un enfant gauche ou droite du grand-parent de z , $p[p[z]]$, ce qui est déterminé en ligne 2. Nous ne donnons le code que pour le cas où $p[z]$ est un enfant gauche. Le nœud $p[p[z]]$ existe, car d'après la partie (b) de l'invariant de boucle, si $p[z]$ est la racine, alors $p[z]$ est noir. Comme on n'entre dans une itération de la boucle que si $p[z]$ est rouge, on sait que $p[z]$ ne peut pas être la racine. Donc, $p[p[z]]$ existe.

Le cas 1 se distingue des cas 2 et 3 par la couleur de l'« oncle » (frère du parent) de z . La ligne 3 fait pointer y vers l'oncle de z , $droite[p[p[z]]]$, et un test est fait en ligne 4. Si y est rouge, alors on est dans le cas 1. Sinon, le contrôle passe aux cas 2 et 3. Dans les trois cas de figure, le grand-parent de z , $p[p[z]]$, est noir ; en effet, le parent $p[z]$ est rouge et la propriété 4 n'est violée qu'entre z et $p[z]$.

► **Cas 1 :** l'oncle de z , y , est rouge

La figure 13.3 montre la situation du cas 1 (lignes 5–8). Le cas 1 se produit quand $p[z]$ et y sont tous les deux rouges. Comme $p[p[z]]$ est noir, on peut colorier $p[z]$ et y en noir, corrigeant ainsi le problème d'avoir z et $p[z]$ rouges tous les deux,

et colorier $p[p[z]]$ en rouge, préservant ainsi la propriété 5. On répète ensuite la boucle **tant que** avec $p[p[z]]$ faisant office de nouveau nœud z . Le pointeur z remonte de deux crans dans l'arbre.

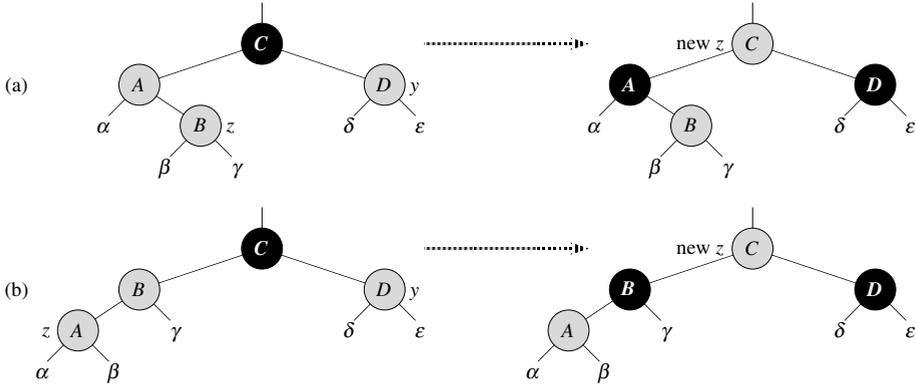


Figure 13.5 Cas 1 de la procédure RN-INSÉRER. La propriété 4 est violée, car z et son parent $p[z]$ sont tous les deux rouges. On effectue la même action, que (a) z soit un enfant droite ou (b) que z soit un enfant gauche. Chacun des sous-arbres $\alpha, \beta, \gamma, \delta$ et ϵ a une racine noire, et chacun a la même hauteur noire. Le code du cas 1 change les couleurs de certains nœuds, pour préserver la propriété 5 : tous les chemins qui descendent d'un nœud vers une feuille ont le même nombre de noirs. La boucle **tant que** continue en prenant le grand-parent du nœud $z, p[p[z]]$, comme nouveau z . Il ne peut maintenant y avoir violation de la propriété 4 qu'entre le nouveau z , qui est rouge, et son parent, si ce dernier est rouge lui aussi.

Nous allons montrer que le cas 1 conserve l'invariant de boucle au début de l'itération suivante. z désignera le nœud z dans l'itération courante, et $z' = p[p[z]]$ désignera le nœud z dans le test fait en ligne 1 lors de l'itération suivante.

- Comme cette itération colorie $p[p[z]]$ en rouge, le nœud z' est rouge au début de l'itération suivante.
- Le nœud $p[z']$ est $p[p[p[z]]]$ dans cette itération, et la couleur de ce nœud ne change pas. Si ce nœud est la racine, il était noir avant cette itération, et il reste noir au début de l'itération suivante.
- Nous avons déjà prouvé que le cas 1 préserve la propriété 5, et il est manifeste qu'il n'introduit pas de violation des propriétés 1 ou 3.

Si le nœud z' est la racine au début de l'itération suivante, alors le cas 1 avait corrigé l'unique violation, celle de la propriété 4, dans l'itération courante. Comme z' est rouge et que c'est la racine, la propriété 2 devient la seule à être violée, et cette violation est due à z' .

Si le nœud z' n'est pas la racine au début de l'itération suivante, alors le cas 1 n'a pas créé de violation de la propriété 2. Le cas 1 avait corrigé l'unique violation, celle de la propriété 4, qui existait au début de l'itération courante. Il avait ensuite

colorié z' en rouge et laissé $p[z']$ inchangé. Si $p[z']$ était noir, il n'y a pas violation de la propriété 4. Si $p[z']$ était rouge, le coloriage de z' en rouge avait créé une seule violation de la propriété 4 entre z' et $p[z']$.

- Cas 2 : l'oncle de z, y , est noir et z est un enfant droite
- Cas 3 : l'oncle de z, y , est noir et z est un enfant gauche

Dans les cas 2 et 3, la couleur de l'oncle de z, y , est le noir. Les deux cas se distinguent par le fait que z est un enfant droite ou un enfant gauche de $p[z]$. Les lignes 10–11 constituent le cas 2, illustré à la figure 13.6 en même temps que le cas 3. Dans le cas 2, le nœud z est un enfant droite de son parent. On utilise immédiatement une rotation gauche pour obtenir le cas 3 (lignes 12–14), dans lequel le nœud z est un enfant gauche. Comme z et $p[z]$ sont tous les deux rouges, la rotation n'affecte ni la hauteur noire des nœuds, ni la propriété 5. Que l'on entre dans le cas 3 directement ou par l'intermédiaire du cas 2, l'oncle de z, y , est noir, car autrement on aurait exécuté le cas 1. De plus, le nœud $p[p[z]]$ existe, car on a démontré que ce nœud existait au moment où les lignes 2 et 3 avaient été exécutées, et après avoir remonté z d'un niveau en ligne 10 puis l'avoir descendu d'un niveau en ligne 11, l'identité de $p[p[z]]$ reste telle quelle. Dans le cas 3, on exécute des changements de couleur et une rotation droite, ce qui préserve la propriété 5 ; ensuite, comme on n'a plus deux nœuds rouges d'affilée, on a fini. Le corps de la boucle **tant que** n'est plus exécuté une nouvelle fois, vu que $p[z]$ est maintenant noir.

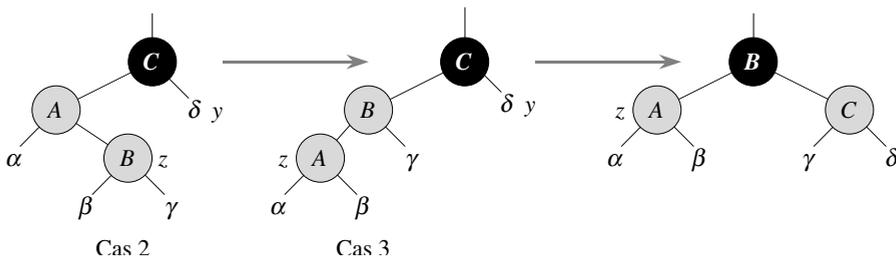


Figure 13.6 Cas 2 et 3 de la procédure RN-INSÉRER. Comme dans le cas 1, la propriété 4 est violée dans le cas 2 ou le cas 3 car z et son parent $p[z]$ sont tous les deux rouges. Chacun des sous-arbres α, β, γ et δ a une racine noire (α, β et γ d'après la propriété 4, et δ car autrement on serait dans le cas 1), et chacun a la même hauteur noire. Le cas 2 devient cas 3 suite à une rotation gauche, ce qui préserve la propriété 5 : tous les chemins qui descendent d'un nœud vers une feuille ont le même nombre de noirs. Le cas 3 entraîne des changements de couleur et une rotation droite, ce qui a pour effet de préserver aussi la propriété 5. La boucle **tant que** se termine ensuite, car la propriété 4 est vérifiée : il n'y a plus deux nœuds rouges d'affilée.

Montrons maintenant que les cas 2 et 3 conservent l'invariant de boucle. (Comme nous venons de le démontrer, $p[z]$ sera noir lors du prochain test fait en ligne 1, et il n'y aura pas d'autre exécution du corps de la boucle.)

- a) Le cas 2 fait pointer z vers $p[z]$, qui est rouge. IL n'y a pas d'autre changement, concernant z ou sa couleur, dans les cas 2 et 3.
- b) Le cas 3 rend $p[z]$ noir de façon que, si $p[z]$ est la racine au début de l'itération suivante, la racine soit noire.
- c) Comme dans le cas 1, les propriétés 1, 3 et 5 sont préservées dans les cas 2 et 3. Comme le nœud z n'est pas la racine dans les cas 2 et 3, on sait qu'il n'y a pas violation de la propriété 2. Les cas 2 et 3 n'introduisent aucune violation de la propriété 2, car le seul nœud qui est colorié en rouge devient un enfant d'un nœud noir suite à la rotation du cas 3.
- Les cas 2 et 3 corrigent l'unique violation, celle de la propriété 4, et ne créent aucune autre violation.

Ayant montré que chaque itération de la boucle conserve l'invariant, nous avons montré que RN-INSÉRER-CORRECTION restaure effectivement les propriétés rouge-noir.

a) Analyse

Quel est le temps d'exécution de RN-INSÉRER ? Comme la hauteur d'un arbre rouge-noir à n nœuds est $O(\lg n)$, les lignes 1–16 de RN-INSÉRER prennent un temps $O(\lg n)$. Dans RN-INSÉRER-CORRECTION, la boucle **tant que** ne se répète que si c'est le cas 1 qui est exécuté, puis le pointeur z remonte de deux niveaux dans l'arbre. Le nombre total de fois que la boucle **tant que** peut être exécutée est donc $O(\lg n)$. Ainsi, RN-INSÉRER demande un temps total de $O(\lg n)$. Il est intéressant de noter qu'elle ne fait jamais plus de deux rotations, car la boucle **tant que** se termine si c'est le cas 2 ou 3 qui est exécuté.

Exercices

13.3.1 A la ligne ?? de RN-INSÉRER, on colorie en rouge le nouveau nœud inséré z . Notez que si nous avons choisi de colorier z en noir, la propriété 4 d'arbre rouge-noir n'aurait pas été violée. Pourquoi donc ne pas avoir choisi de colorier z en noir ?

13.3.2 Montrer les arbres rouge-noir qui résultent de l'insertion successive des clés 41, 38, 31, 12, 19, 8 dans un arbre rouge-noir initialement vide.

13.3.3 On suppose que la hauteur noire de chacun des sous-arbres $\alpha, \beta, \gamma, \delta, \varepsilon$ des figures 13.5 et 13.6 vaut k . Étiqueter chaque nœud, sur chaque figure, par sa hauteur noire pour vérifier que la propriété 5 est préservée par la transformation indiquée.

13.3.4 Le professeur Muche se fait du souci : il se dit que RN-INSÉRER-CORRECTION risque d'affecter à $\text{couleur}[\text{nil}[T]]$ la couleur ROUGE, auquel cas le test fait en ligne 1 ne forcerait pas la boucle à se terminer quand z est la racine. Montrer que le professeur se fait du souci pour rien, en prouvant que RN-INSÉRER-CORRECTION ne met jamais dans $\text{couleur}[\text{nil}[T]]$ la valeur ROUGE.

13.3.5 Soit un arbre rouge-noir formé par l'insertion de n nœuds *via* RN-INSÉRER. Montrer que, si $n > 1$, l'arbre a au moins un nœud rouge.

13.3.6 Dire comment on pourrait implémenter RN-INSÉRER efficacement si la représentation des arbres rouge-noir n'imposait pas de stocker des pointeurs vers parent.

13.4 SUPPRESSION

Comme les autres opérations primitives sur un arbre rouge-noir à n nœuds, la suppression d'un nœud requiert un temps $O(\lg n)$. Supprimer un nœud d'un arbre rouge-noir est cependant légèrement plus compliqué que d'en insérer un.

La procédure RN-SUPPRIMER est une version légèrement modifiée de la procédure ARBRE-SUPPRIMER (section 12.3). Après suppression d'un nœud, elle appelle une procédure auxiliaire RN-SUPPRIMER-CORRECTION qui modifie des couleurs et fait des rotations pour restaurer les propriétés rouge-noir.

```

RN-SUPPRIMER( $T, z$ )
1  si gauche[ $z$ ] = nil[ $T$ ] ou droite[ $z$ ] = nil[ $T$ ]
2    alors  $y \leftarrow z$ 
3    sinon  $y \leftarrow$  ARBRE-SUCESSEUR( $z$ )
4  si gauche[ $y$ ]  $\neq$  nil[ $T$ ]
5    alors  $x \leftarrow$  gauche[ $y$ ]
6    sinon  $x \leftarrow$  droite[ $y$ ]
7   $p[x] \leftarrow p[y]$ 
8  si  $p[y] = \text{nil}[T]$ 
9    alors racine[ $T$ ]  $\leftarrow x$ 
10 sinon si  $y =$  gauche[ $p[y]$ ]
11     alors gauche[ $p[y]$ ]  $\leftarrow x$ 
12     sinon droite[ $p[y]$ ]  $\leftarrow x$ 
13 si  $y \neq z$ 
14   alors clé[ $z$ ]  $\leftarrow$  clé[ $y$ ]
15     copier données satellite de  $y$  dans  $z$ 
16 si couleur[ $y$ ] = NOIR
17   alors RN-SUPPRIMER-CORRECTION( $T, x$ )
18 retourner  $y$ 

```

Il y a trois différences entre les procédures ARBRE-SUPPRIMER et RN-SUPPRIMER. Primo, toutes les références à NIL dans ARBRE-SUPPRIMER sont remplacées par des références à la sentinelle $\text{nil}[T]$ dans RN-SUPPRIMER. Secundo, le test déterminant si x est NIL en ligne 7 de ARBRE-SUPPRIMER a été enlevé, et l'affectation $p[x] \leftarrow p[y]$ est devenue inconditionnelle en ligne 7 de RN-SUPPRIMER. Donc, si x est la sentinelle $\text{nil}[T]$, son pointeur de parent pointe vers le parent du nœud supprimé y . Tertio, un appel à RN-SUPPRIMER-CORRECTION est fait aux lignes 16–17 si y est noir. Si

y est rouge, les propriétés rouge-noir restent vérifiées après suppression de y , et ce pour les raisons suivantes :

- aucune hauteur noire n’a été modifiée dans l’arbre,
- il n’y a pas eu apparition de nœuds rouges adjacents, et
- comme y n’aurait pas pu être la racine s’il était rouge, la racine reste noire.

Le nœud x passé à RN-SUPPRIMER-CORRECTION est l’un ou l’autre des deux nœuds suivants : le nœud qui était l’unique enfant de y avant suppression de y si y avait un enfant qui n’était pas la sentinelle $nil[T]$, ou, si y n’avait pas d’enfants, x est la sentinelle $nil[T]$. Dans ce dernier cas, l’assignation inconditionnelle de la ligne 7 garantit que le parent de x est maintenant le nœud qui était précédemment le parent de y , que x soit un nœud interne porteur de clé ou qu’il soit la sentinelle $nil[T]$.

On peut maintenant regarder comment la procédure RN-SUPPRIMER-CORRECTION restaure les propriétés rouge-noir de l’arbre.

RN-SUPPRIMER-CORRECTION(T, x)

```

1  tant que  $x \neq \text{racine}[T]$  et  $\text{couleur}[x] = \text{NOIR}$ 
2    faire si  $x = \text{gauche}[p[x]]$ 
3      alors  $w \leftarrow \text{droite}[p[x]]$ 
4        si  $\text{couleur}[w] = \text{ROUGE}$ 
5          alors  $\text{couleur}[w] \leftarrow \text{NOIR}$                                 ▷ Cas 1
6               $\text{couleur}[p[x]] \leftarrow \text{ROUGE}$                             ▷ Cas 1
7              ROTATION-GAUCHE( $T, p[x]$ )                                ▷ Cas 1
8               $w \leftarrow \text{droite}[p[x]]$                                 ▷ Cas 1
9        si  $\text{couleur}[\text{gauche}[w]] = \text{NOIR}$  et  $\text{couleur}[\text{droite}[w]] = \text{NOIR}$ 
10       alors  $\text{couleur}[w] \leftarrow \text{ROUGE}$                                 ▷ Cas 2
11            $x \leftarrow p[x]$                                             ▷ Cas 2
12       sinon si  $\text{couleur}[\text{droite}[w]] = \text{NOIR}$ 
13         alors  $\text{couleur}[\text{gauche}[w]] \leftarrow \text{NOIR}$                     ▷ Cas 3
14              $\text{couleur}[w] \leftarrow \text{ROUGE}$                                 ▷ Cas 3
15             ROTATION-DROITE( $T, w$ )                                    ▷ Cas 3
16              $w \leftarrow \text{droite}[p[x]]$                                 ▷ Cas 3
17              $\text{couleur}[w] \leftarrow \text{couleur}[p[x]]$                         ▷ Cas 4
18              $\text{couleur}[p[x]] \leftarrow \text{NOIR}$                                 ▷ Cas 4
19              $\text{couleur}[\text{droite}[w]] \leftarrow \text{NOIR}$                         ▷ Cas 4
20             ROTATION-GAUCHE( $T, p[x]$ )                                ▷ Cas 4
21              $x \leftarrow \text{racine}[T]$                                     ▷ Cas 4
22       sinon (idem clause alors avec « droite » et « gauche » échangés)
23      $\text{couleur}[x] \leftarrow \text{NOIR}$ 

```

Si le nœud supprimé y dans RN-SUPPRIMER est noir, il peut se produire trois problèmes. Primo, si y était la racine et qu’un enfant rouge de y devient la nouvelle racine, on enfreint la propriété 2. Secundo, si x et $p[y]$ (qui maintenant est aussi $p[x]$)

étaient tous les deux rouges, alors on a enfreint la propriété 4. Tertio, la suppression de y entraîne que chaque chemin qui contenait y se retrouve avec un nœud noir en moins. Donc, la propriété 5 est désormais enfreinte par tous les ancêtres de y dans l'arbre. On peut corriger ce problème en disant que le nœud x a un noir « supplémentaire ». C'est-à-dire, si on ajoute 1 au compteur de nœuds noirs de chaque chemin contenant x , alors moyennant cette interprétation, la propriété 5 tient toujours. Quand on supprime le nœud noir y , on « pousse » sa noirceur sur son enfant. Le problème est que maintenant le nœud x n'est ni rouge ni noir, ce qui est une violation de la propriété 1. À la place, le nœud x est « doublement noir » ou « rouge-et-noir » ; selon le cas, il contribue pour 2 ou 1 au compteur de nœuds noirs d'un chemin contenant x . L'attribut *couleur* de x reste ROUGE (si x est rouge-et-noir) ou NOIR (si x est doublement noir). Autrement dit, le noir supplémentaire d'un nœud se reflète au niveau du pointage de x vers le nœud plutôt qu'au niveau de l'attribut *couleur*.

La procédure RN-SUPPRIMER-CORRECTION restaure les propriétés 1, 2 et 4. Les exercices 13.4.1 et 13.4.2 vous demanderont de montrer que la procédure restaure les propriétés 2 et 4 ; donc, dans le reste de cette section, nous nous concentrerons sur la propriété 1. Le but de la boucle **tant que** des lignes 1–22 est de faire remonter le noir supplémentaire dans l'arbre jusqu'à ce que

- 1) x pointe vers un nœud rouge-et-noir, auquel cas on colorie x en noir (simple) en ligne 23,
- 2) x pointe vers la racine, auquel cas on peut se contenter de « supprimer » le noir en trop, ou
- 3) on peut procéder à des rotations et recoloriages idoines.

Au sein de la boucle **tant que**, x pointe toujours vers un nœud doublement noir qui n'est pas la racine. On détermine en ligne 2 si x est un enfant gauche ou un enfant droite de son parent $p[x]$. (Nous avons donné le code pour le cas où x est un enfant gauche ; le cas où x est un enfant droite, en ligne 22, est symétrique.) On gère un pointeur w vers le frère de x . Comme le nœud x est doublement noir, le nœud w ne peut pas être *nil*[T] ; sinon, le nombre de noirs du chemin allant de $p[x]$ à la feuille (noire simple) w serait inférieur au nombre du chemin reliant $p[x]$ à x .

Les quatre cas ⁽²⁾ du code sont illustrés à la figure 13.7. Avant d'examiner chaque cas en détail, voyons de manière plus générale comment on peut vérifier que la transformation dans chacun des cas préserve la propriété 5. L'idée fondamentale est que, dans chaque cas, le nombre de nœuds noirs (dont le noir supplémentaire de x) partant de (et incluant) la racine du sous-arbre dessiné pour chacun des sous-arbres $\alpha, \beta, \dots, \zeta$ est préservé par la transformation. Donc, si la propriété 5 est vérifiée avant la transformation, elle l'est après. Par exemple, sur la figure 13.7(a), qui illustre le cas 1, le nombre de nœuds noirs entre la racine et l'un des sous-arbres α ou β vaut 3, avant comme après la transformation. (Rappelons une fois de plus que le

(2) Comme dans RN-INSÉRER-CORRECTION, les cas de RN-SUPPRIMER-CORRECTION ne sont pas mutuellement exclusifs.

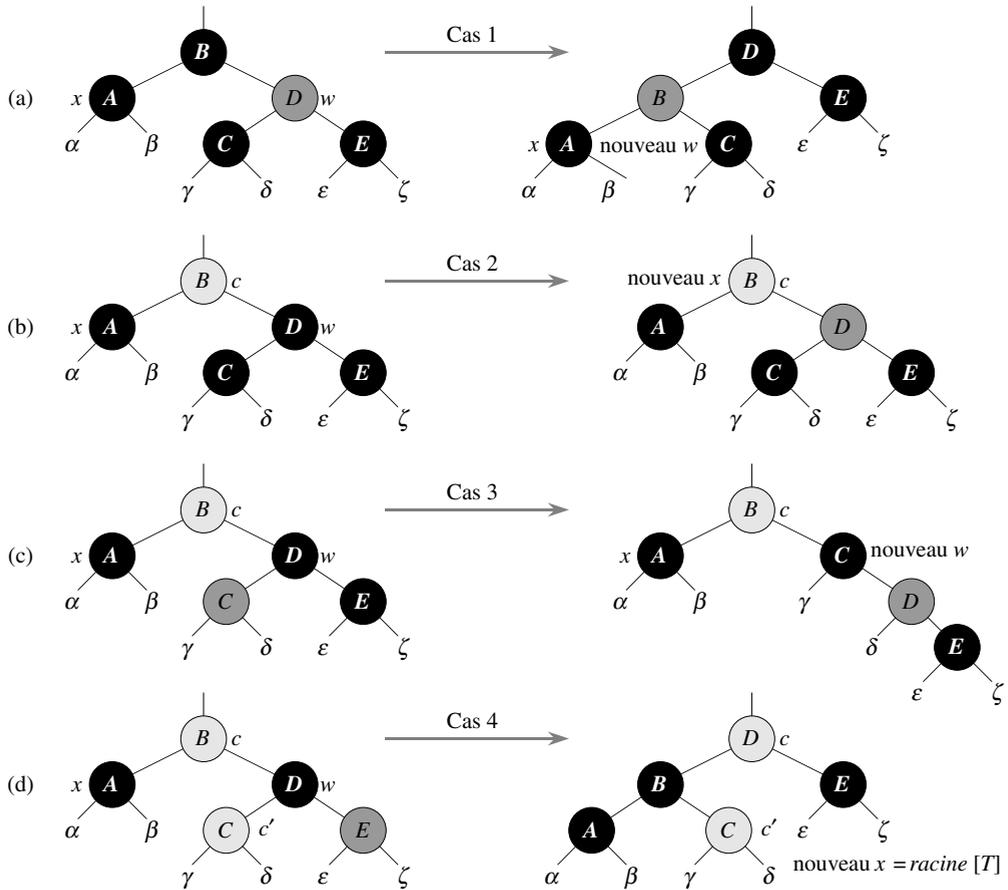


Figure 13.7 Les cas dans la boucle **tant que** de la procédure **RN-SUPPRIMER-CORRECTION**. Les nœuds en noir ont des attributs *couleur* valant NOIR, les nœuds en gris foncé ont des attributs *couleur* valant ROUGE et les nœuds en gris clair ont des attributs *couleur*, représentés par c et c' , qui peuvent valoir ROUGE ou NOIR. Les lettres $\alpha, \beta, \dots, \zeta$ représentent des sous-arbres arbitraires. Dans chaque cas, la configuration de gauche est transformée en la configuration de droite *via* modification de certaines couleurs et/ou exécution d'une rotation. Chaque nœud pointé par x a un noir supplémentaire et est doublement noir ou rouge-et-noir. Le seul cas qui provoque la répétition de la boucle est le cas 2. **(a)** Le cas 1 est transformé en cas 2, 3 ou 4 *via* permutation des couleurs des nœuds B et D et exécution d'une rotation gauche. **(b)** Dans le cas 2, le noir supplémentaire représenté par le pointeur x est déplacé vers le haut *via* coloriage du nœud D en rouge et configuration de x pour le faire pointer vers le nœud B . Si l'on entre dans le cas 2 *via* le cas 1, la boucle **tant que** se termine car le nouveau nœud x est rouge-et-noir, et donc la valeur c de son attribut *couleur* est ROUGE. **(c)** Le cas 3 est transformé en cas 4 *via* permutation des couleurs des nœuds C et D et exécution d'une rotation droite. **(d)** Dans le cas 4, le noir supplémentaire représenté par x peut être supprimé *via* modification de certaines couleurs et exécution d'une rotation gauche (sans violation des propriétés rouge-noir), et la boucle se termine.

nœud x compte pour un noir de plus.) De même, le nombre de nœuds noirs entre la racine et l'un des arbres γ , δ , ε et ζ vaut 2, avant comme après la transformation. À la figure 13.7(b), le comptage doit tenir compte de la valeur c de l'attribut *couleur* de la racine du sous-arbre affiché, qui peut être ROUGE ou NOIR. Si l'on définit $\text{count}(\text{ROUGE}) = 0$ et $\text{count}(\text{NOIR}) = 1$, alors le nombre de nœuds noirs entre la racine et α est $2 + \text{count}(c)$, avant comme après la transformation. Dans ce cas, après la transformation, le nouveau nœud x a l'attribut *couleur* c , mais ce nœud est soit rouge-et-noir (si $c = \text{ROUGE}$) ou doublement noir (si $c = \text{NOIR}$). On peut vérifier les autres cas de façon similaire (voir exercice 13.4.5).

► Cas 1 : le frère de x , w , est rouge

Le cas 1 (lignes 5–8 de RN-SUPPRIMER-CORRECTION et figure 13.7(a)) se produit quand le nœud w , le frère de x , est rouge. Comme w doit avoir des enfants noirs, on peut permuter les couleurs de w et $p[x]$ puis effectuer une rotation gauche sur $p[x]$ sans enfreindre l'une des propriétés rouge-noir. Le nouveau frère de x , qui est l'un des enfants de w avant la rotation, est désormais noir, et donc le cas 1 est ramené au cas 2, 3 ou 4.

Les cas 2, 3 et 4 se produisent quand le nœud w est noir ; ces cas se distinguent par les couleurs des enfants de w .

► Cas 2 : le frère de x , w , est noir, et les deux enfants de w sont noirs

Dans le cas 2 (lignes 10–11 de RN-SUPPRIMER-CORRECTION et figure 13.7(b)), les deux enfants de w sont noirs. Comme w est noir lui aussi, on enlève un noir à x et à w , ce qui laisse à x un seul noir et qui laisse w rouge. Pour compenser la suppression d'un noir dans x et w , on voudrait ajouter un noir supplémentaire à $p[x]$, qui était au départ rouge ou noir. Pour ce faire, on répète la boucle **tant que** en faisant de $p[x]$ le nouveau nœud x . Observez que, si l'on entre dans le cas 2 *via* le cas 1, le nouveau nœud x est rouge-et-noir, vu que le $p[x]$ originel était rouge. Donc, la valeur c de l'attribut *couleur* du nouveau nœud x est ROUGE, et la boucle se termine quand elle teste la condition de boucle. Le nouveau nœud x est alors colorié en noir (simple), en ligne 23.

► Cas 3 : le frère de x , w , est noir, l'enfant gauche de w est rouge et l'enfant droite de w est noir

Le cas 3 (lignes 13–16 et figure 13.7(c)) se produit quand w est noir, son enfant gauche est rouge et son enfant droite est noir. On peut permuter les couleurs de w et de son enfant gauche $gauche[w]$ puis faire une rotation droite sur w sans enfreindre les propriétés rouge-noir. Le nouveau frère w de x est maintenant un nœud noir avec un enfant droite rouge, et le cas 3 est donc ramené au cas 4.

► Cas 4 : le frère de x , w , est noir et l'enfant droite de w est rouge

Le cas 4 (lignes 17–21 et figure 13.7(d)) se produit quand le frère du nœud x est noir et que l'enfant droite de w est rouge. En faisant des modifications de couleur et en

effectuant une rotation gauche sur $p[x]$, on peut supprimer le noir en trop de x , rendant ce dernier noir simple, sans violer de propriété rouge-noir. Faire de x la racine forcera la boucle **tant que** à se terminer quand elle testera la condition de bouclage.

a) Analyse

Quel est le temps d'exécution de RN-SUPPRIMER ? Comme la hauteur d'un arbre rouge-noir à n nœuds est $O(\lg n)$, le coût total de la procédure, hors appel à RN-SUPPRIMER-CORRECTION, est $O(\lg n)$. À l'intérieur de RN-SUPPRIMER-CORRECTION, les cas 1, 3 et 4 se terminent chacun après avoir effectué un nombre constant de changements de couleur et au plus trois rotations. Le cas 2 est le seul pour lequel la boucle **tant que** peut se répéter ; dans ce cas, le pointeur x remonte dans l'arbre au plus $O(\lg n)$ fois, et aucune rotation n'est effectuée. La procédure RN-SUPPRIMER-CORRECTION prend donc un temps $O(\lg n)$ et effectue au plus trois rotations ; le coût total de RN-SUPPRIMER est donc également $O(\lg n)$.

Exercices

13.4.1 Montrer que, après exécution de RN-SUPPRIMER-CORRECTION, la racine de l'arbre est forcément noire.

13.4.2 Montrer que, si dans RN-SUPPRIMER x et $p[y]$ sont rouges, alors la propriété 4 est restaurée par l'appel RN-SUPPRIMER-CORRECTION(T, x).

13.4.3 Dans l'exercice 13.3.2, on vous demandait quel était l'arbre issu de l'insertion successive des clés 41, 38, 31, 12, 19, 8 dans un arbre initialement vide. Montrer à présent les arbres rouge-noir issus de la suppression successive des clés 8, 12, 19, 31, 38, 41.

13.4.4 Dans quelles lignes du code de RN-SUPPRIMER-CORRECTION pourrait-on examiner ou modifier la sentinelle $nil[T]$?

13.4.5 Pour chacun des cas de la figure 13.7, donner la quantité de nœuds noirs entre la racine du sous-arbre représenté et chacun des sous-arbres $\alpha, \beta, \dots, \zeta$, et vérifier que chaque quantité reste la même après la transformation. Lorsqu'un nœud a l'attribut de couleur c ou c' , utiliser la notation $\text{compteur}(c)$ ou $\text{compteur}(c')$ symboliquement dans votre décompte.

13.4.6 Les professeurs Picrate et Vinasse ont peur qu'au début du cas 1 de RN-SUPPRIMER-CORRECTION, le nœud $p[x]$ puisse ne pas être noir. Si les professeurs pensent juste, alors les lignes 5–6 sont erronées. Montrer que $p[x]$ est forcément noir au début du cas 1, de sorte que les professeurs se font du souci pour rien.

13.4.7 Supposons qu'un nœud x soit inséré dans un arbre rouge-noir *via* RN-INSÉRER, puis immédiatement supprimé *via* RN-SUPPRIMER. L'arbre rouge-noir résultant est-il le même que l'arbre initial ? Justifier la réponse.

PROBLÈMES

13.1. Ensembles dynamiques persistants

Au cours d'un algorithme, on a parfois besoin de conserver d'anciennes versions d'un ensemble dynamique quand celui-ci est modifié. Un tel ensemble est dit *persistant*. Une façon d'implémenter un ensemble persistant est de copier l'ensemble tout entier chaque fois qu'il est modifié, mais cette approche peut ralentir un programme et également consommer beaucoup d'espace. On peut parfois faire beaucoup mieux.

Considérons un ensemble persistant S avec les opérations INSÉRER, SUPPRIMER et RECHERCHER, qu'on implémente à l'aide des arbres binaires de recherche représentés sur la figure 13.8(a). On conserve une nouvelle racine pour chaque version de l'ensemble. Pour insérer la clé 5 dans l'ensemble, on crée un nouveau nœud de clé 5. Ce nœud devient l'enfant gauche d'un nouveau nœud de clé 7, puisqu'on ne peut pas modifier le nœud de clé 7 existant. De même, le nouveau nœud de clé 7 devient l'enfant gauche d'un nouveau nœud de clé 8 dont l'enfant droit est l'ancien nœud de clé 10. Le nouveau nœud de clé 8 devient, à son tour, l'enfant droit d'un nouvelle racine r' de clé 4 dont l'enfant gauche est l'ancien nœud de clé 3. On ne recopie ainsi qu'une partie de l'arbre et l'on partage certains nœuds avec l'arbre initial, comme le montre la figure 13.8(b).

On suppose que chaque nœud de l'arbre contient les champs *clé*, *gauche* et *droite* mais aucun champ parent. (Voir aussi exercice 13.3.6.)

- a. Pour un arbre binaire de recherche persistant général, identifier les nœuds qui ont besoin d'être modifiés pour insérer une clé k ou supprimer un nœud y .
- b. Écrire une procédure ARBRE-PERSISTANT-INSÉRER qui, étant donné un arbre persistant T et une clé k à insérer, retourne un nouvel arbre persistant T' qui résulte de l'insertion de k dans T .
- c. Si la hauteur de l'arbre binaire de recherche persistant T est h , quels sont les besoins en temps et en espace de votre implémentation de ARBRE-PERSISTANT-INSÉRER ? (L'espace nécessaire est proportionnel au nombre de nouveaux nœuds alloués.)
- d. Supposons qu'on ait inclus le champ parent dans chaque nœud. Dans ce cas, ARBRE-PERSISTANT-INSÉRER doit effectuer des recopies supplémentaires. Démontrer que les besoins en temps et en espace de ARBRE-PERSISTANT-INSÉRER sont alors $\Omega(n)$, où n est le nombre de nœuds dans l'arbre.
- e. Montrer comment utiliser des arbres rouge-noir pour garantir que le temps d'exécution et l'espace nécessaire, dans le cas le plus défavorable, sont $O(\lg n)$ par insertion ou suppression.

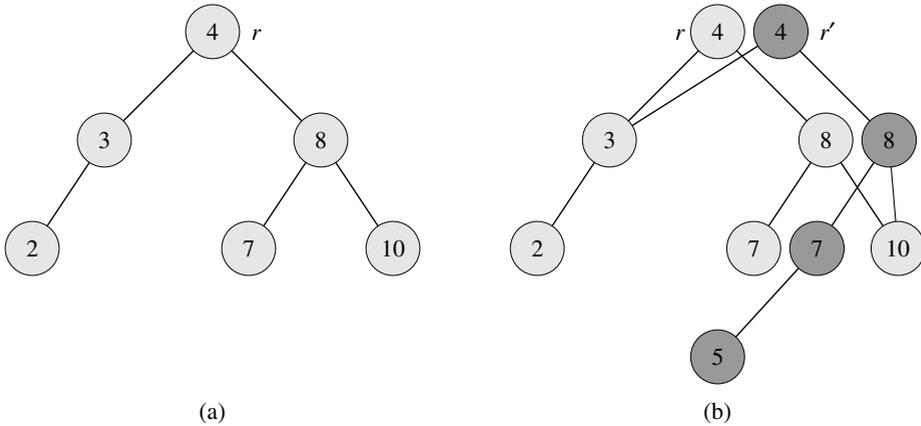


Figure 13.8 (a) Un arbre binaire de recherche avec les clés 2, 3, 4, 7, 8, 10. (b) L'arbre binaire de recherche persistant qui résulte de l'insertion de la clé 5. La version la plus récente de l'ensemble est constituée des nœuds accessibles depuis la racine r' et la précédente version est constituée des nœuds accessibles depuis r . Les nœuds en gris foncé sont ajoutés lorsque la clé 5 est insérée.

13.2. Jointure d'arbres rouge-noir

L'opération de *jointure* prend deux ensembles dynamiques S_1 et S_2 et un élément x tels que, pour tous $x_1 \in S_1$ et $x_2 \in S_2$, on ait $clé[x_1] \leq clé[x] \leq clé[x_2]$. Elle retourne un ensemble $S = S_1 \cup \{x\} \cup S_2$. Dans ce problème, on s'intéresse à l'implémentation de l'opération de jointure sur les arbres rouge-noir.

- a.** Étant donné un arbre rouge-noir T , on décide de conserver sa hauteur noire dans un champ $hn[T]$. Montrer que ce champ peut être mis à jour par RN-INSÉRER et RN-SUPPRIMER sans que l'on ait besoin de gérer de l'espace en plus dans les nœuds de l'arbre, ni d'augmenter les temps d'exécution asymptotiques. Montrer que, lors d'une descente le long de T , on peut déterminer la hauteur noire de chaque nœud visité en temps $O(1)$ par nœud visité.

On souhaite implémenter l'opération $RN-JOINTURE(T_1, x, T_2)$, qui détruit T_1 et T_2 et retourne un arbre rouge-noir $T = T_1 \cup \{x\} \cup T_2$. Soit n le nombre total de nœuds dans T_1 et T_2 .

- b.** On suppose que $hn[T_1] \geq hn[T_2]$. Décrire un algorithme à temps $O(\lg n)$ qui trouve un nœud noir y dans T_1 ayant une clé plus grande que tous les nœuds de hauteur noire $hn[T_2]$.
- c.** Soit T_y le sous-arbre enraciné en y . Décrire comment remplacer T_y par $T_y \cup \{x\} \cup T_2$ en temps $O(1)$ sans détruire la propriété d'arbre binaire de recherche.
- d.** Quelle couleur faut-il donner à x pour que les propriétés d'arbre rouge-noir 1, 3 et 5 soient conservées ? Décrire comment restaurer les propriétés 2 et 4 en temps $O(\lg n)$.

- e. Prouver que l'hypothèse faite en (b) ne nuit pas à la généralité. Décrire la situation symétrique qui se produit quand $hn[T_1] \leq hn[T_2]$.
- f. Montrer que le temps d'exécution de RN-JOINTURE est $O(\lg n)$.

13.3. Arbres AVL

Un *arbre AVL* est un arbre binaire *équilibré en hauteur* : pour chaque nœud x , les hauteurs des sous-arbres gauche et droite de x ne diffèrent que de 1 au plus. Pour implémenter un arbre AVL, on gère un champ supplémentaire dans chaque nœud : $h[x]$ est la hauteur du nœud x . Comme pour tout autre arbre binaire de recherche T , on suppose que $racine[T]$ pointe vers le nœud racine.

- a. Prouver qu'un arbre AVL à n nœuds a une hauteur $O(\lg n)$. (*Conseil* : Prouver que, dans un arbre AVL de hauteur h , il y a au moins F_h nœuds, où F_h est le h ème nombre de Fibonacci.)
- b. Pour faire une insertion dans un arbre AVL, on commence par placer un nœud à l'endroit approprié dans l'ordre de l'arbre binaire. Après cette insertion, l'arbre risque de ne plus être équilibré en hauteur. Plus précisément, les hauteurs des enfants gauche et droite d'un certain nœud risquent de différer de 2. Décrire une procédure EQUILIBRE(x) qui prend en entrée un sous-arbre enraciné en x dont les enfants gauche et droite sont équilibrés en hauteur et ont des hauteurs qui diffèrent d'au plus 2 (c'est-à-dire, $|h[droite[x]] - h[gauche[x]]| \leq 2$), puis qui modifie le sous-arbre enraciné en x pour qu'il soit équilibré en hauteur. (*Conseil* : Employer des rotations.)
- c. En utilisant la partie (b), décrire une procédure récursive AVL-INSÉRER(x, z) qui prend en entrée un nœud x d'un arbre AVL et un nœud nouvellement créé z (dont la clé a déjà été initialisée), puis qui ajoute z au sous-arbre enraciné en x en préservant la propriété selon laquelle x est la racine d'un arbre AVL. Comme dans ARBRE-INSÉRER de la section 12.3, on supposera que $clé[z]$ a déjà été remplie et que $gauche[z] = \text{NIL}$ et $droite[z] = \text{NIL}$; on supposera également que $h[z] = 0$. Donc, pour insérer le nœud z dans l'arbre AVL T , on appelle AVL-INSÉRER($racine[T], z$).
- d. Donner un exemple d'arbre AVL à n nœuds dans lequel une opération AVL-INSÉRER entraîne l'exécution de $\Omega(\lg n)$ rotations.

13.4. Arbretas

Quand on insère un ensemble de n éléments dans un arbre binaire de recherche, l'arbre résultant risque d'être abominablement déséquilibré et d'entraîner des temps de recherche très longs. Cependant, comme nous l'avons vu à la section 12.4, les arbres binaires construits aléatoirement ont tendance à être équilibrés. Par conséquent, une stratégie qui, en moyenne, construit un arbre équilibré pour un ensemble fixé d'éléments consiste à permuter aléatoirement les éléments, puis à les insérer dans cet ordre dans l'arbre.

Mais si l'on n'a pas tous les éléments en même temps ? Si on reçoit les éléments un par un, est-ce que l'on peut encore construire un arbre binaire à partir de ces éléments ?

Nous allons examiner une structure de données qui répond à ce problème. Un *arbretas* est un arbre binaire de recherche qui a une façon modifiée d'ordonner les nœuds. La figure 13.9 montre un exemple. Comme d'habitude, chaque nœud x de l'arbre a une valeur de clé $clé[x]$. En outre, on affecte $priorité[x]$ qui est un nombre aléatoire choisi de manière indépendante pour chaque nœud. On suppose que toutes les priorités sont distinctes et aussi que toutes les clés sont distinctes. Les nœuds de l'arbretas sont ordonnés de telle façon que les clés respectent la propriété d'arbre binaire de recherche et que les priorités respectent la propriété d'ordre de tas min :

- Si v est un enfant gauche de u , alors $clé[v] < clé[u]$.
- Si v est un enfant droite de u , alors $clé[v] > clé[u]$.
- Si v est un enfant de u , alors $priorité[v] > priorité[u]$.

(La combinaison de ces propriétés explique le nom « arbretas » : c'est un mélange d'arbre (tree) et de tas (heap).)

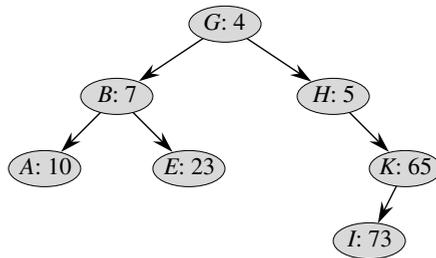


Figure 13.9 Un arbretas. Chaque nœud x est étiqueté par $clé[x]$: $priorité[x]$. Par exemple, la racine a la clé G et la priorité 4.

Voici une façon utile de se représenter les arbretas. Supposons que l'on insère les nœuds x_1, x_2, \dots, x_n , avec les clés associées, dans un arbretas. Alors, l'arbretas résultant est l'arbre qui aurait été formé si les nœuds avaient été insérés dans un arbre binaire classique dans l'ordre donné par leurs priorités (choisies au hasard) ; c'est-à-dire, $priorité[x_i] < priorité[x_j]$ signifie que x_i a été inséré avant x_j .

- a. Montrer que, étant donné un ensemble de nœuds x_1, x_2, \dots, x_n avec les clés et priorités associées (toutes distinctes), il existe un unique arbretas qui soit associé à ces nœuds.
- b. Montrer que la hauteur attendue d'un arbretas est $\Theta(\lg n)$, et donc que le temps de recherche d'une valeur dans l'arbretas est $\Theta(\lg n)$.

Voyons comment insérer un nœud dans un arbretas existant. La première chose que nous faisons, c'est d'assigner au nouveau nœud une priorité aléatoire. On appelle ensuite l'algorithme d'insertion, baptisé ARBRETAS-INSÉRER, dont le fonctionnement est illustré par la figure 13.10.

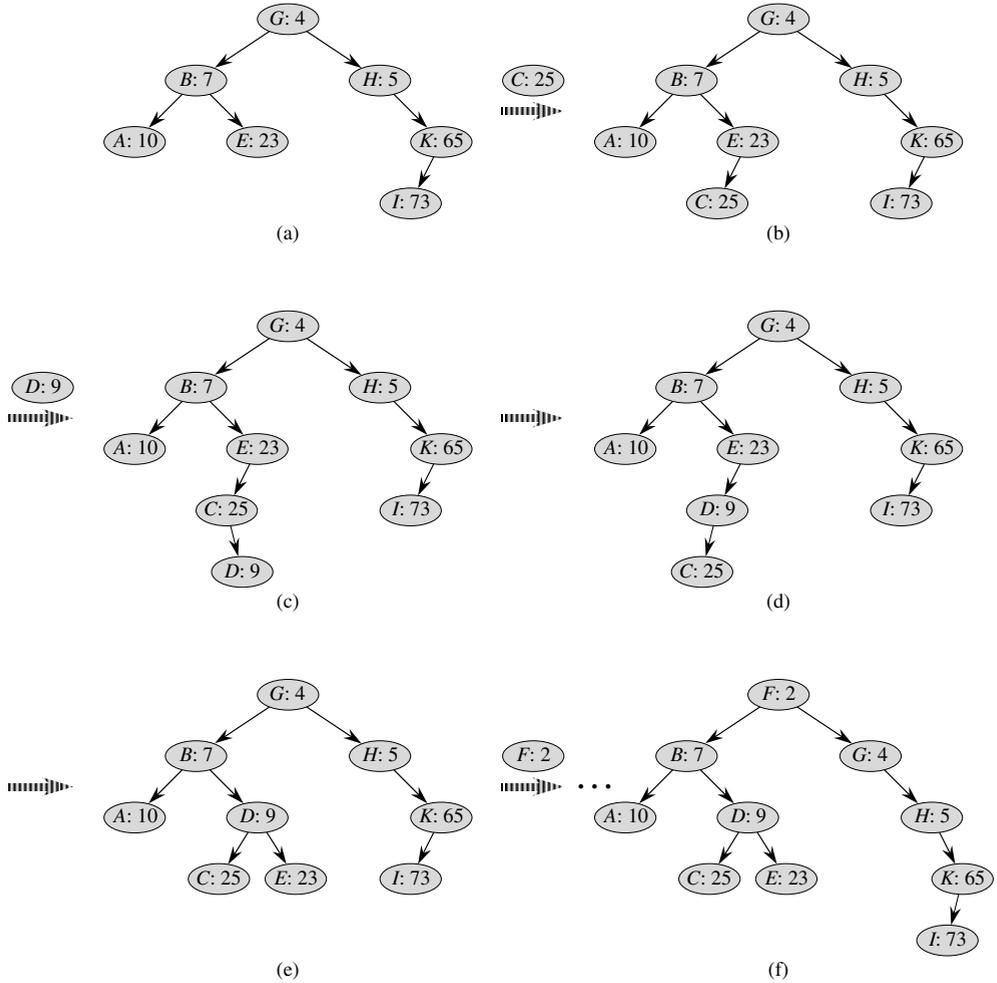


Figure 13.10 Fonctionnement de ARBRETAS-INSÉRER. **(a)** L'arbretas original, avant l'insertion. **(b)** L'arbretas après insertion d'un nœud de clé C et de priorité 25. **(c)–(d)** Étapes intermédiaires de l'insertion d'un nœud de clé D et de priorité 9. **(e)** L'arbretas après les insertions faites aux parties (c) et (d). **(f)** L'arbretas après insertion d'un nœud de clé F et de priorité 2.

- c.** Expliquer le fonctionnement de ARBRETAS-INSÉRER. Expliquer le concept en langage naturel, puis donner le pseudo code. (*Conseil* : On exécute la procédure classique d'insertion dans un arbre binaire de recherche, puis on fait des rotations pour restaurer la propriété d'ordre de tas min.)
- d.** Montrer que le temps d'exécution attendu de ARBRETAS-INSÉRER est $\Theta(\lg n)$.

ARBRETAS-INSÉRER effectue une recherche, puis une suite de rotations. Bien que ces deux opérations aient le même temps d'exécution attendu, elles ont des coûts différents en pratique. Une recherche lit des données dans l'arbrétas sans le modifier. En comparaison, une rotation modifie des pointeurs vers parent et vers enfant dans l'arbrétas. Sur la plupart des ordinateurs, les lectures sont plus rapides que les écritures. On voudrait donc que ARBRETAS-INSÉRER fasse peu de rotations. Nous allons montrer que le nombre attendu de rotations effectuées est borné par une constante.

Pour ce faire, on a besoin de quelques définitions, qui sont illustrées sur la figure 13.11. La *dorsale gauche* d'un arbre binaire de recherche T est le chemin reliant la racine au nœud qui a la clé minimale. Autrement dit, la dorsale gauche est le chemin partant de la racine et composé uniquement de bords gauches. De manière symétrique, la *dorsale droite* de T est le chemin partant de la racine et composé uniquement de bords droits. La *longueur* d'une dorsale est le nombre de nœuds qu'elle contient.

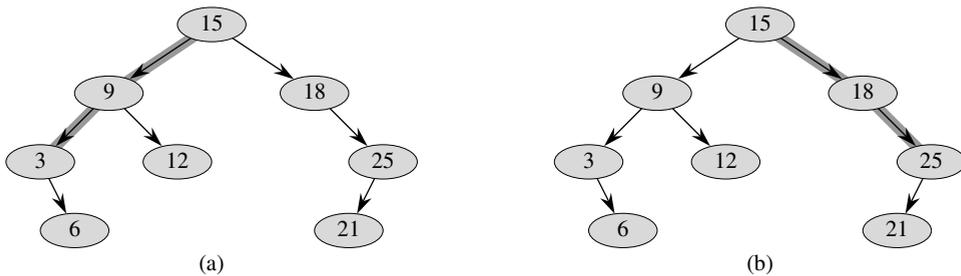


Figure 13.11 Dorsales d'un arbre binaire de recherche. La dorsale gauche est dessinée sur fond ombré en (a), et la dorsale droite est dessinée sur fond ombré en (b).

- e. On considère l'arbrétas T juste après l'insertion de x via ARBRETAS-INSÉRER. Soit C la longueur de la dorsale droite du sous-arbre gauche de x . Soit D la longueur de la dorsale gauche du sous-arbre droit de x . Prouver que le nombre total de rotations effectuées lors de l'insertion de x est égal à $C + D$.

On va maintenant calculer les valeurs attendues de C et D . Sans nuire à la généralité, on suppose que les clés sont $1, 2, \dots, n$, vu que l'on ne fait que les comparer entre elles.

Pour les nœuds x et y , où $y \neq x$, soient $k = \text{clé}[x]$ et $i = \text{clé}[y]$, et définissons les variables indicatrices

$$X_{i,k} = \mathbb{I} \{y \text{ est dans la dorsale droite du sous-arbre gauche de } x \text{ (dans } T)\} .$$

- f. Montrer que $X_{i,k} = 1$ si et seulement si $\text{priorité}[y] > \text{priorité}[x]$, $\text{clé}[y] < \text{clé}[x]$ et, pour tout z tel que $\text{clé}[y] < \text{clé}[z] < \text{clé}[x]$, on a $\text{priorité}[y] < \text{priorité}[z]$.

g. Montrer que

$$\Pr \{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

h. Montrer que

$$E[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

i. Utiliser un argument de symétrie pour montrer que

$$E[D] = 1 - \frac{1}{n-k+1}.$$

j. En conclure que le nombre attendu de rotations effectuées lors de l'insertion d'un nœud dans un arbretas est inférieur à 2.

NOTES

Le principe d'équilibrage d'un arbre de recherche est dû à Adel'son-Vel'skiï et Landis [2], qui introduisirent en 1962 une classe d'arbres équilibrés appelée « arbres AVL » (décrite au problème 13-3). Une autre classe d'arbres de recherche, dite « arbres 2-3 », fut introduite par J. E. Hopcroft (non publié) en 1970. L'équilibre est maintenu *via* manipulation des degrés des nœuds dans l'arbre. Une généralisation des arbres 2-3, présentée par Bayer et McCreight [32] et appelée B-arbres, forme le sujet du chapitre 18.

Les arbres rouge-noir ont été inventés par Bayer [31] sous le nom « B-arbres binaires symétriques ». Guibas et Sedgewick [135] ont étudié leur propriétés en détail et introduit la convention du coloriage rouge-noir. Andersson [15] donne une variante à code plus simple des arbres rouge-noir. Weiss [311] appelle cette variante arbres AA. Un arbre AA ressemble à un arbre rouge-noir, sauf que les enfants de gauche ne peuvent jamais être rouges.

Les arbretas ont été proposés par Seidel et Aragon [271]. C'est l'implémentation par défaut employée pour un dictionnaire dans LEDA, collection bien implémentée de structures de données et d'algorithmes.

Il existe une foule d'autres variantes pour arbres binaires équilibrés, dont : arbres équilibrés en poids [230], arbres k -voisin [213] et arbres bouc-émissaire [108].

La variante la plus étonnante est peut-être celle des « arbres déployés », introduits par Sleator et Tarjan [281], qui sont des arbres « auto-adaptatifs ». (Tarjan [292] fournit une bonne description des arbres déployés). Ces arbres maintiennent leur équilibre sans aucune condition explicite d'équilibrage, comme par exemple la couleur. À la place, des « opérations de déploiement » (mettant en jeu des rotations) sont faites dans l'arbre chaque fois qu'on y accède. Le coût amorti (voir chapitre 17) de chaque opération sur un arbre à n nœuds est $O(\lg n)$.

Les listes à saut (skip list) [251] sont une solution de remplacement pour les arbres binaires équilibrés. Une liste à saut est une liste chaînée, dotée d'un certain nombre de pointeurs supplémentaires. Chaque opération de dictionnaire se fait avec un temps moyen de $O(\lg n)$ sur une liste à saut de n éléments.

Chapitre 14

Extension d'une structure de données

Pour faire face à certaines situations pratiques, il suffit parfois d'une structure de données classique, par exemple une liste doublement chaînée, une table de hachage ou un arbre binaire de recherche ; mais d'autres contextes demandent un peu d'imagination. Cependant, les cas où l'on doit créer de toutes pièces un nouveau type de structure de données sont rares. Le plus souvent, il suffit d'étendre une structure classique en y stockant des informations supplémentaires. On peut alors programmer de nouvelles opérations qui permettent à la structure de données de supporter l'application voulue. Toutefois, étendre une structure de données n'est pas toujours évident, car les informations supplémentaires doivent être mises à jour et prises en compte par les opérations ordinaires sur la structure de données.

Ce chapitre étudie deux structures de données qui sont construites en étendant des arbres rouge-noir. La section 14.1 décrit une structure de données qui supporte les opérations générales sur les rangs dans un ensemble dynamique. On peut alors trouver rapidement le i ème plus petit nombre d'un ensemble ou le rang d'un élément donné dans un ensemble totalement ordonné. La section 14.2 modélise le processus d'extension d'une structure de données et fournit un théorème qui peut simplifier l'extension des arbres rouge-noir. Dans la section 14.3, on s'aidera de ce théorème pour concevoir une structure de données capable de gérer un ensemble dynamique d'intervalles, par exemple des intervalles de temps. Étant donné un intervalle entre deux requêtes, on peut alors rapidement trouver dans l'ensemble un intervalle avec lequel il se superpose.

14.1 RANGS DYNAMIQUES

Le chapitre 9 a introduit la notion de rang. Plus précisément, le i ème rang d'un ensemble à n éléments, où $i \in \{1, 2, \dots, n\}$, est tout simplement l'élément de l'ensemble ayant la i ème plus petite clé. Nous avons vu qu'on pouvait retrouver un rang quelconque en $O(n)$ dans un ensemble non trié. Dans cette section, on verra comment les arbres rouge-noir peuvent être modifiés pour qu'on puisse retrouver un rang quelconque en $O(\lg n)$. Nous verrons également comment déterminer le **rang** d'un élément, à savoir sa position dans l'ordre linéaire de l'ensemble dans un temps $O(\lg n)$.

La figure 14.1 montre une structure de données qui permet d'exécuter rapidement les opérations sur les rangs. Un **arbre de rangs** T est tout simplement un arbre rouge-noir avec des informations supplémentaires conservées à chaque nœud. En plus des champs habituels utilisés dans un arbre rouge-noir, $clé[x]$, $couleur[x]$, $p[x]$, $gauche[x]$, et $droite[x]$, tout nœud x contient aussi un champ $taille[x]$. Ce champ contient le nombre de nœuds (internes) du sous-arbre enraciné en x (x compris), autrement dit la taille du sous-arbre. Si l'on suppose que la taille de la sentinelle est 0, c'est-à-dire si l'on affecte $taille[NIL] = 0$, alors on a l'identité

$$taille[x] = taille[gauche[x]] + taille[droite[x]] + 1 .$$

Nous n'imposons pas aux clés d'un arbre de rangs qu'elles soient distinctes. (Par exemple, l'arbre de la figure 14.1 a deux clés de valeur 14 et deux clés de valeur 21.) En présence de clés identiques, la notion précédente de rang n'est pas bien définie. On supprime cette ambiguïté, pour un arbre de rangs, en définissant le rang d'un élément comme étant la position à laquelle il serait affiché dans le cadre d'un parcours infixe de l'arbre. Sur la figure 14.1, par exemple, la clé 14 stockée dans un nœud noir a le rang 5, et la clé 14 stockée dans un nœud rouge a le rang 6.

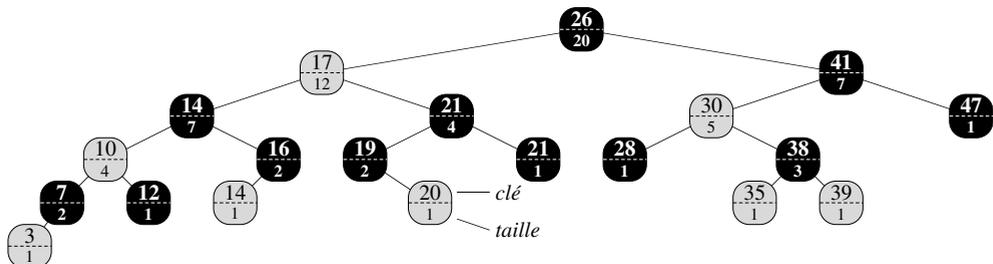


Figure 14.1 Un arbre de rangs, qui est un arbre rouge-noir étendu. Les nœuds rouges sont en gris, les nœuds noirs sont en noir. En plus des champs habituels, chaque nœud x comporte un champ $taille[x]$, qui représente le nombre de nœuds du sous-arbre enraciné en x .

a) Retrouver un élément d'un rang donné

Avant de montrer comment on peut maintenir cette nouvelle information à jour pendant l'insertion et la suppression, examinons l'implémentation de deux requêtes sur les rangs qui utilisent cette information supplémentaire. On commence par une opération qui récupère un élément d'un rang donné. La procédure `RÉCUPÉRER-RANG(x, i)` retourne un pointeur sur le nœud contenant la i ème plus petite clé du sous-arbre enraciné en x . Pour trouver la i ème plus petite clé dans un arbre de rangs T , on appelle `RÉCUPÉRER-RANG($racine[T], i$)`.

Le principe de `RÉCUPÉRER-RANG` ressemble à celui des algorithmes de sélection du chapitre 9. La valeur de `taille[gauche[x]]` est le nombre de nœuds qui précèdent x dans un parcours infixe du sous-arbre enraciné en x . Donc `taille[gauche[x]] + 1` est le rang de x dans le sous-arbre enraciné en x .

```

RÉCUPÉRER-RANG( $x, i$ )
1   $r \leftarrow \text{taille}[\text{gauche}[x]] + 1$ 
2  si  $i = r$ 
3    alors retourner  $x$ 
4  sinon si  $i < r$ 
5    alors retourner RÉCUPÉRER-RANG(gauche[ $x$ ],  $i$ )
6  sinon retourner RÉCUPÉRER-RANG(droite[ $x$ ],  $i - r$ )

```

A la ligne 1 de `RÉCUPÉRER-RANG`, on calcule r , rang du nœud x à l'intérieur du sous-arbre enraciné en x . Si $i = r$, alors le nœud x est le i ème plus petit élément, et on retourne x à la ligne 3. Si $i < r$, alors le i ème plus petit élément se trouve dans le sous-arbre gauche de x , et l'on commence une récursivité dans `gauche[x]` à la ligne 5. Si $i > r$, le i ème plus petit élément se trouve dans le sous-arbre droit de x . Comme il y a r éléments dans le sous-arbre enraciné en x qui précèdent le sous-arbre droit de x lors du parcours infixe, le i ème plus petit élément du sous-arbre enraciné en x est le $(i - r)$ ème plus petit élément du sous-arbre enraciné en `droite[x]`. Cet élément est déterminé récursivement à la ligne 6.

Pour comprendre le comportement de `RÉCUPÉRER-RANG`, considérons la recherche du 17ème plus petit élément de l'arbre de rangs de la figure 14.1. On commence avec x à la racine, donc la clé est 26, et avec $i = 17$. Comme la taille du sous-arbre gauche de 26 est 12, son rang est 13. On sait donc que le nœud de rang 17 est le $17 - 13 = 4$ ème plus petit élément dans le sous-arbre droit de 26. Après l'appel récursif, x est le nœud de clé 41, et $i = 4$. Comme la taille du sous-arbre gauche de 41 est 5, son rang à l'intérieur du sous-arbre est 6. On sait donc que le nœud de rang 4 est le 4ème plus petit élément du sous-arbre gauche de 41. Après l'appel récursif, x pointe sur le nœud de clé 30, et son rang à l'intérieur de son sous-arbre est 2. On continue donc la récursivité pour trouver le $4 - 2 = 2$ ème plus petit élément du sous-arbre dont la racine est le nœud de clé 38. On voit à présent que son sous-arbre gauche à une taille 1, ce qui signifie que c'est le deuxième plus petit élément. Un pointeur sur le nœud de clé 38 est alors retourné par la procédure.

Comme chaque appel récursif descend d'un niveau dans l'arbre de rangs, le temps total de RÉCUPÉRER-RANG est au pire proportionnel à la hauteur de l'arbre. Comme c'est un arbre rouge-noir, sa hauteur est $O(\lg n)$, où n est le nombre de nœuds. Le temps d'exécution de RÉCUPÉRER-RANG est donc $O(\lg n)$ pour un ensemble dynamique à n éléments.

b) Déterminer le rang d'un élément

Étant donné un pointeur vers un nœud x d'un arbre de rangs T , la procédure DÉTERMINER-RANG retourne la position de x dans l'ordre linéaire tel que déterminé dans un parcours infixe de T .

```

DÉTERMINER-RANG( $T, x$ )
1   $r \leftarrow \text{taille}[\text{gauche}[x]] + 1$ 
2   $y \leftarrow x$ 
3  tant que  $y \neq \text{racine}[T]$ 
4      faire si  $y = \text{droite}[p[y]]$ 
5          alors  $r \leftarrow r + \text{taille}[\text{gauche}[p[y]]] + 1$ 
6           $y \leftarrow p[y]$ 
7  retourner  $r$ 

```

La procédure fonctionne de la manière suivante. Le rang de x peut être vu comme étant le nombre de nœuds qui précèdent x dans un parcours infixe de l'arbre, nombre augmenté de 1 pour tenir compte de x lui-même. DÉTERMINER-RANG conserve l'invariant de boucle que voici : Au début de chaque itération de la boucle **tant que** des lignes 3–6, r est le rang de $\text{clé}[x]$ dans le sous-arbre issu du nœud y .

On va utiliser cet invariant pour montrer que DÉTERMINER-RANG fonctionne correctement :

Initialisation : Avant la première itération, la ligne 1 fait de r le rang de $\text{clé}[x]$ dans le sous-arbre issu de x . L'affectation $y \leftarrow x$ en ligne 2 rend l'invariant vrai lors de la première exécution du test de la ligne 3.

Maintenance : à la fin de chaque itération de la boucle **tant que**, on fait $y \leftarrow p[y]$. Il faut donc montrer que, si r est le rang de $\text{clé}[x]$ dans le sous-arbre issu de y au début du corps de la boucle, alors r est le rang de $\text{clé}[x]$ dans le sous-arbre issu de $p[y]$ à la fin du corps de la boucle. À chaque itération de la boucle **tant que**, on considère le sous-arbre issu de $p[y]$. On a déjà compté le nombre de nœuds du sous-arbre issu de y qui précèdent x dans un parcours infixe, de sorte qu'il faut ajouter les nœuds du sous-arbre issu du frère de y qui précèdent x dans un parcours infixe, plus 1 pour $p[y]$ s'il précède, lui aussi, x . Si y est un enfant gauche, alors ni $p[y]$ ni aucun nœud du sous-arbre droit de $p[y]$ ne précède x , de sorte que l'on ne modifie pas r . Sinon, y est un enfant droit et tous les nœuds du sous-arbre gauche de $p[y]$ précèdent x , ce que fait $p[y]$ lui-même. Donc, en ligne 5, on ajoute $\text{taille}[\text{gauche}[p[y]]] + 1$ à la valeur courante de r .

Terminaison : La boucle se termine quand $y = \text{racine}[T]$, de sorte que le sous-arbre issu de y n'est autre que l'arbre complet. Par conséquent, la valeur de r est le rang de $\text{clé}[x]$ dans l'arbre tout entier.

Comme exemple, lorsqu'on exécute DÉTERMINER-RANG sur l'arbre de rangs de la figure 14.1 pour trouver le rang du nœud de clé 38, on obtient la séquence suivante de valeurs pour $\text{clé}[y]$ et r au sommet de la boucle **tant que** :

itération	$\text{clé}[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

La procédure retourne le rang 17.

Comme chaque itération de la boucle **tant que** prend $O(1)$, et que y remonte d'un niveau dans l'arbre à chaque itération, le temps d'exécution de DÉTERMINER-RANG est au pire proportionnel à la hauteur de l'arbre : $O(\lg n)$ sur un arbre de rangs à n nœuds.

c) Conserver la taille des sous-arbres

Connaissant le champ *taille* de chaque nœud, RÉCUPÉRER-RANG et DÉTERMINER-RANG peuvent rapidement calculer les informations concernant les rangs. Mais si ces champs ne peuvent pas être pris en compte efficacement par les opérations de modification élémentaires des arbres rouge-noir, notre travail n'aura servi à rien. Nous allons montrer que les tailles des sous-arbres peuvent être gérées aussi bien lors de l'insertion que lors de la suppression, sans affecter le temps d'exécution asymptotique de chaque opération.

Nous avons remarqué à la section 13.3 que l'insertion dans un arbre rouge-noir était constitués de deux opérations distinctes. La première phase descendait le long de l'arbre à partir de la racine, pour insérer le nouveau nœud comme fils d'un nœud existant. La seconde phase remontait le long l'arbre en changeant les couleurs, et terminait en effectuant des rotations permettant de préserver les propriétés des arbres rouge-noir.

Pour gérer la taille des sous-arbres pendant la première phase, on se contente d'incrémenter $\text{taille}[x]$ pour chaque nœud x du chemin reliant la racine aux feuilles. Le nouveau nœud se voit affecter une *taille* égale à 1. Comme le chemin parcouru contient $O(\lg n)$ nœuds, le coût supplémentaire requis par la gestion des champs *taille* est $O(\lg n)$.

Lors de la seconde phase, les seuls changements structurels affectant l'arbre rouge-noir sont causés par les rotations, qui sont au plus au nombre de deux. Par ailleurs, une rotation est une opération locale : elle ne remet en question que les champs *taille* des deux nœuds de chaque côté du lien autour duquel la rotation s'effectue. En

partant du code de $\text{ROTATION-GAUCHE}(T, x)$ à la section 13.2, on ajoute les lignes suivantes :

```

13  taille[y] ← taille[x]
14  taille[x] ← taille[gauche[x]] + taille[droite[x]] + 1

```

La figure 14.2 illustre la mise à jour des champs. La modification apportée à ROTATION-DROITE est symétrique.

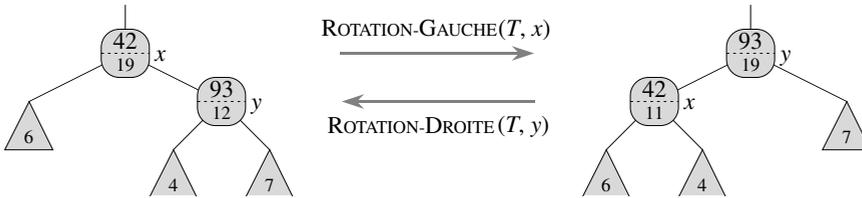


Figure 14.2 Mise à jour de la taille des sous-arbres lors des rotations. Les champs *taille* qui doivent être mis à jour sont ceux des nœuds situés de part et d'autre du lien autour duquel s'effectue la rotation. Les mises à jour sont locales, et ne demandent que les informations *taille* conservées dans x , y et dans les racines des sous-arbres représentés par des triangles.

Comme au plus deux rotations sont effectuées pendant l'insertion dans un arbre rouge-noir, on ne dépense qu'un temps supplémentaire en $O(1)$ pour mettre à jour les champs *taille* lors de la seconde phase. Le temps total pris par l'insertion dans un arbre de rangs à n nœuds est donc $O(\lg n)$, ce qui est asymptotiquement identique à celui d'un arbre rouge-noir ordinaire.

La suppression dans un arbre rouge-noir est également constituée de deux phases : la première agit sur l'arbre de recherche sous-jacent, et la seconde génère au plus trois rotations, sans provoquer d'autres modifications structurelles. (Voir section 13.4.) La première phase détache un nœud y . Pour mettre à jour la taille des sous-arbres, on se contente de longer un chemin remontant depuis le nœud y vers la racine, en décrémentant le champ *taille* de chaque nœud situé sur le chemin. Comme ce chemin a la longueur $O(\lg n)$ dans un arbre rouge-noir à n nœuds, le temps supplémentaire requis pour maintenir les champs *taille* lors de la première phase est $O(\lg n)$. Les rotations en $O(1)$ de la seconde phase de la suppression peuvent être gérées de la même façon que pour l'insertion. Par conséquent, l'insertion comme la suppression, maintenance des champs *taille* comprise, prennent un temps $O(\lg n)$ pour un arbre de rangs à n nœuds.

Exercices

14.1.1 Montrer comment $\text{RÉCUPÉRER-RANG}(T, 10)$ agit sur l'arbre rouge-noir T de la figure 14.1.

14.1.2 Montrer l'action de DÉTERMINER-RANG(T, x) sur l'arbre rouge-noir T de la figure 14.1 et le nœud x pour lequel $clé[x] = 35$.

14.1.3 Écrire une version non récursive de RÉCUPÉRER-RANG.

14.1.4 Écrire une procédure récursive RECHERCHER-RANG-CLÉ(T, k) qui prend en entrée un arbre de rangs T et une clé k , et qui retourne le rang de k dans l'ensemble dynamique représenté par T . On supposera que les clés de T sont distinctes.

14.1.5 Étant donné un élément x dans un arbre de rangs à n nœuds, et un entier naturel i , comment peut-on déterminer le i ème successeur de x dans l'ordre linéaire de l'arbre avec un temps $O(\lg n)$?

14.1.6 Observons que, chaque fois que le champ *taille* est utilisé dans RÉCUPÉRER-RANG ou DÉTERMINER-RANG, il ne sert qu'à calculer le rang du nœud dans le sous-arbre issu de ce nœud. Supposons donc que l'on décide de stocker dans chaque nœud son rang dans le sous-arbre dont il est la racine. Montrer comment gérer cette donnée lors de l'insertion et de la suppression. (Ne pas oublier que ces deux opérations peuvent provoquer des rotations).

14.1.7 Montrer comment utiliser un arbre de rangs pour compter en temps $O(n \lg n)$ le nombre d'inversions (voir problème 2.4) dans un tableau de taille n .

14.1.8 * Considérons n cordes dans un cercle, chacune étant définie par ses extrémités. Décrire un algorithme en $O(n \lg n)$ permettant de déterminer le nombre de paires de cordes qui se croisent à l'intérieur du cercle. (Par exemple, si les n cordes sont toutes des diamètres qui se croisent au centre, la bonne réponse sera $\binom{n}{2}$.) On supposera que deux cordes quelconques n'ont jamais d'extrémité commune.

14.2 COMMENT ÉTENDRE UNE STRUCTURE DE DONNÉES

En phase de conception d'un algorithme, on a très fréquemment recours à l'extension d'une structure de données classique pour supporter des fonctionnalités supplémentaires. Nous y ferons à nouveau appel dans la prochaine section pour concevoir une structure de données permettant de gérer des intervalles. Dans cette section, nous examinerons les étapes qui président à cette extension. Nous démontrerons également un théorème nous permettant dans de nombreux cas d'étendre facilement des arbres rouge-noir.

L'extension d'une structure de données peut être divisée en quatre étapes :

- 1) choisir une structure de données sous-jacente,
- 2) déterminer les informations supplémentaires à gérer dans la structure sous-jacente.
- 3) vérifier que les informations supplémentaires seront compatibles avec les opérations habituelles de modification de la structure de données sous-jacente, et
- 4) créer de nouvelles opérations.

Comme avec toute méthode de conception normative, vous ne devrez pas suivre aveuglément ces étapes dans l'ordre. Tout travail de conception contient le plus souvent une partie d'essais et d'erreurs, et se déroule habituellement en parallèle sur toutes les étapes. Il ne sert à rien, par exemple, de déterminer les informations supplémentaires et de développer les nouvelles opérations (étapes 2 et 4) si l'on est incapable de gérer efficacement les informations supplémentaires. Néanmoins, ce plan en quatre étapes balise bien vos efforts d'extension d'une structure de données, et c'est également un bon moyen d'organiser la documentation d'une structure de données étendue.

Nous avons suivi ces quatre étapes dans la section 14.1 pour établir nos arbres de rangs. Pour l'étape 1, nous avons choisi des arbres rouge-noir comme structure de données principale. Un indice de la pertinence des arbres rouge-noir est donné par leur support efficace d'autres opérations d'ordre total sur des ensembles dynamiques, comme MINIMUM, MAXIMUM, SUCCESSEUR et PRÉDÉCESSEUR.

Pour l'étape 2, nous avons fourni le champ *taille*, qui stocke dans chaque nœud x la taille du sous-arbre de racine x . En général, les informations supplémentaires rendent les opérations plus efficaces. Par exemple, on aurait pu implémenter RÉCUPÉRER-RANG et DÉTERMINER-RANG en se servant uniquement des clés conservées dans l'arbre, mais elles ne se seraient pas exécutées en $O(\lg n)$. Parfois, l'information additionnelle est un pointeur et non une donnée, comme dans l'exercice 14.2.1.

Dans l'étape 3, nous nous sommes assurés que l'insertion et la suppression pouvaient gérer les champs *taille* tout en continuant à s'exécuter en $O(\lg n)$. Idéalement, un petit nombre de modifications de la structure de données devrait suffire à gérer les informations supplémentaires. Par exemple, si l'on s'était contenté de stocker dans chaque nœud son rang dans l'arbre, les procédures RÉCUPÉRER-RANG et DÉTERMINER-RANG s'exécuteraient rapidement, mais l'insertion d'un nouvel élément minimal provoquerait le changement de cette donnée dans tous les nœuds de l'arbre. En revanche, quand ce sont les tailles de sous-arbre qui sont stockées, l'insertion d'un nouvel élément provoque des changements dans seulement $O(\lg n)$ nœuds.

Pour l'étape 4, nous avons développé les opérations RÉCUPÉRER-RANG et DÉTERMINER-RANG. Après tout, c'est le besoin de disposer de nouvelles opérations qui nous a fait étendre la structure de données. Parfois, au lieu de développer de nouvelles opérations, on se sert de l'information supplémentaire pour rendre plus efficaces des opérations déjà existantes, comme dans l'exercice 14.2.1.

a) Extension des arbres rouge-noir

Lorsque des arbres rouge-noir forment le squelette d'une structure étendue, on peut démontrer que certains types d'informations supplémentaires peuvent toujours être gérés efficacement par l'insertion et la suppression, ce qui facilite grandement l'étape 3. La démonstration du théorème suivant est similaire à la démonstration qui, à la section 14.1, prouve que le champ *taille* peut être géré pour des arbres de rangs.

Théorème 14.1 (Extension d'un arbre rouge-noir) *Soit c un champ qui étend la structure d'un arbre rouge-noir T à n nœuds ; on suppose que le contenu de c pour un nœud x peut être calculé seulement à l'aide des informations présentes dans les nœuds x , $\text{gauche}[x]$ et $\text{droite}[x]$, y compris $c[\text{gauche}[x]]$ et $c[\text{droite}[x]]$. Alors, il est possible de gérer les valeurs de c dans tous les nœuds de T lors d'une insertion ou d'une suppression sans affecter asymptotiquement le temps $O(\lg n)$ de ces opérations.*

Démonstration : L'idée générale de la démonstration est qu'un changement dans le champ c d'un nœud x ne se propage qu'aux ancêtres de x dans l'arbre. Autrement dit, modifier $c[x]$ peut obliger à mettre à jour $c[p[x]]$, Mais rien d'autre ; mettre à jour $c[p[x]]$ peut obliger à modifier $c[p[p[x]]]$, mais rien d'autre ; et ainsi de suite, jusqu'à la racine. Lorsque $c[\text{racine}[T]]$ est mis à jour, aucun autre nœud ne dépend de la nouvelle valeur, et donc le processus se termine. Comme la hauteur d'un arbre rouge-noir est $O(\lg n)$, modifier le champ c d'un nœud a un coût $O(\lg n)$ nécessaire à la mise à jour des nœuds qui dépendent de cette modification.

L'insertion d'un nœud x dans T est constituée de deux phases. (Voir la section 13.3.) Pendant la première phase, x est inséré comme fils d'un nœud $p[x]$ déjà présent. La valeur de $c[x]$ peut être calculée en un temps $O(1)$ puisque, par hypothèse, elle ne dépend que des informations présentes dans les autres champs de x et des données contenues dans les fils de x ; mais les enfants de x sont tous deux la sentinelle $\text{nil}[T]$. Une fois que $c[x]$ est calculé, la modification remonte dans l'arbre. Le temps total de la première phase d'insertion est donc $O(\lg n)$. Pendant la deuxième phase, les seuls changements structurels apportés à l'arbre viennent des rotations. Comme deux nœuds seulement changent lors d'une rotation, le temps global de mise à jour des champs c est $O(\lg n)$ par rotation. Comme le nombre de rotations pendant l'insertion est au plus égal à deux, le temps total de l'insertion est $O(\lg n)$.

Comme l'insertion, la suppression comprend deux phases. (Voir la section 13.4.) Lors de la première phase, des modifications surviennent si le nœud supprimé est remplacé par son successeur, et quand le nœud supprimé ou son successeur est détaché. La propagation des mises à jours de c provoquées par ces modifications coûte au plus $O(\lg n)$, puisque les changements modifient l'arbre localement. La correction d'un arbre rouge-noir pendant la seconde phase fait appel à trois rotations au plus, et chaque rotation requiert au plus un temps $O(\lg n)$ pour propager les mises à jour sur c . Ainsi, comme pour l'insertion, le temps total de la suppression est $O(\lg n)$. \square

Dans de nombreux cas, par exemple pour la gestion des champs *taille* dans les arbres de rangs, le coût d'une mise à jour après une rotation est $O(1)$, et non le $O(\lg n)$ trouvé dans la démonstration du théorème 14.1. Un exemple en est donné à l'exercice 14.2.4.

Exercices

14.2.1 Montrer comment mettre en œuvre les requêtes d'ensemble dynamique MINIMUM, MAXIMUM, SUCCESSEUR et PRÉDÉCESSEUR en temps $O(1)$, dans le cas le plus défavorable, sur un arbre de rangs étendu. Les performances asymptotiques des autres opérations sur les arbres de rangs ne devront pas être affectées. (*Conseil :* Ajouter des pointeurs aux nœuds.)

14.2.2 Les hauteurs noires des nœuds d'un arbre rouge-noir peuvent-elles être gérées en tant que champs des nœuds de l'arbre sans que les performances asymptotiques des opérations d'arbre rouge-noir soient affectées ? Justifier la réponse.

14.2.3 Peut-on utiliser un champ supplémentaire dans les nœuds d'un arbre rouge-noir pour gérer efficacement la profondeur des nœuds. Dire pourquoi.

14.2.4 * Soit \otimes un opérateur binaire associatif, et soit a un champ géré dans chaque nœud d'un arbre rouge-noir. Supposons qu'on veuille inclure dans chaque nœud x un champ supplémentaire c tel que $c[x] = a[x_1] \otimes a[x_2] \otimes \dots \otimes a[x_m]$, où x_1, x_2, \dots, x_m est la liste infixée des nœuds du sous-arbre enraciné en x . Montrer que les champs c peuvent être correctement mis à jour en $O(1)$ après une rotation. Modifier légèrement votre démonstration pour montrer que les champs *taille* des arbres de rangs peuvent être gérés en temps $O(1)$ par rotation.

14.2.5 * On souhaite étendre les arbres rouge-noir avec une opération $\text{RN-ENUMÉRER}(x, a, b)$ qui affiche toutes les clés k telles que $a \leq k \leq b$ d'un arbre rouge-noir enraciné en x . Décrire comment RN-ENUMÉRER peut être implémentée en temps $\Theta(m + \lg n)$, où m est le nombre de clés qui sont affichées et n le nombre de nœuds internes de l'arbre. (*Conseil* : il est inutile d'ajouter de nouveaux champs à l'arbre rouge-noir.)

14.3 ARBRES D'INTERVALLES

Dans cette section, on étendra des arbres rouge-noir pour supporter les opérations sur des ensembles dynamiques d'intervalles. Un *intervalle fermé* est une paire ordonnée de nombres réels $[t_1, t_2]$, avec $t_1 \leq t_2$. L'intervalle $[t_1, t_2]$ représente l'ensemble $\{t \in \mathbf{R} : t_1 \leq t \leq t_2\}$. Les intervalles *semi-ouvert* et *ouverts* excluent respectivement de l'ensemble l'une ou leurs deux extrémités. Dans cette section, nous admettrons que les intervalles sont fermés ; étendre les résultats à des intervalles ouverts ou semi-ouverts ne pose conceptuellement aucun problème.

Les intervalles sont pratiques pour représenter des événements qui occupent chacun une période de temps continue. On pourrait, par exemple, avoir envie d'interroger une base de données d'intervalles de temps pour retrouver les événements qui se produisent pendant un intervalle donné. La structure de données de cette section fournit un moyen efficace pour gérer une telle base d'intervalles.

Un intervalle $[t_1, t_2]$ peut être représenté par un objet i , comportant deux champs : $\text{début}[i] = t_1$ et $\text{fin}[i] = t_2$. On dit que des intervalles i et i' se *recoupent* si $i \cap i' \neq \emptyset$, c'est-à-dire si $\text{début}[i] \leq \text{fin}[i']$ et $\text{début}[i'] \leq \text{fin}[i]$. Deux intervalles i et i' quelconques vérifient la *trichotomie d'intervalle*, à savoir qu'une propriété et une seule parmi les trois suivantes est satisfaite :

- a) i et i' se recoupent,
- b) i est à la gauche de i' (c'est-à-dire, $\text{fin}[i] < \text{début}[i']$),
- c) i est à la droite de i' (c'est-à-dire, $\text{fin}[i'] < \text{début}[i]$).

La figure 14.3 illustre ces trois possibilités.

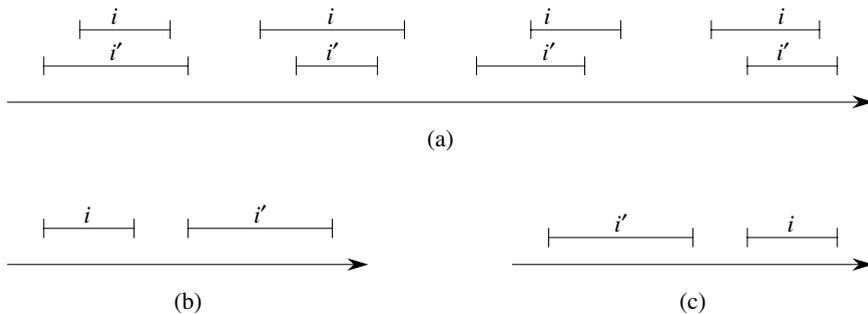


Figure 14.3 La trichotomie d'intervalle pour deux intervalles fermés i et i' . (a) Si i et i' se recoupent, quatre cas se présentent ; pour chacun, $début[i] \leq fin[i']$ et $début[i'] \leq fin[i]$. (b) Les intervalles ne se recoupent pas, et $fin[i] < début[i']$. (c) Les intervalles ne se recoupent pas, et $fin[i'] < début[i]$.

Un **arbre d'intervalles** est un arbre rouge-noir qui gère un ensemble dynamique d'éléments, chaque élément x contenant un intervalle $int[x]$. Les arbres d'intervalles autorisent les opérations suivantes :

INSÉRER-INTERVALLE(T, x) ajoute l'élément x , dont on suppose que le champ int contient un intervalle, à l'arbre d'intervalles T .

SUPPRIMER-INTERVALLE(T, x) ôte l'élément x de l'arbre d'intervalles T .

RECHERCHER-INTERVALLE(T, i) retourne un pointeur sur un élément x de l'arbre d'intervalles T tel que $int[x]$ recoupe l'intervalle i , ou la sentinelle $nil[T]$ si un tel élément n'existe pas dans l'ensemble.

La figure 14.4 montre comment un ensemble d'intervalles peut être représenté par un arbre d'intervalles. Le plan en quatre étapes de la section 14.2 va nous guider pendant que nous reprendrons la conception d'un arbre d'intervalles et les opérations qui s'y rattachent.

a) Étape 1 : Structure de données sous-jacente

On choisit un arbre rouge-noir dans lequel chaque nœud x contient un intervalle $int[x]$ et dont la clé représente l'extrémité gauche, $début[int[x]]$. Ainsi, par un parcours préfixe de la structure de données, on peut recenser les intervalles triés selon leur extrémité gauche.

b) Étape 2 : Information supplémentaire

En plus des intervalles eux-mêmes, chaque nœud x contient une valeur $max[x]$, qui représente la valeur maximale parmi les extrémités d'intervalles stockés dans le sous-arbre enraciné en x .

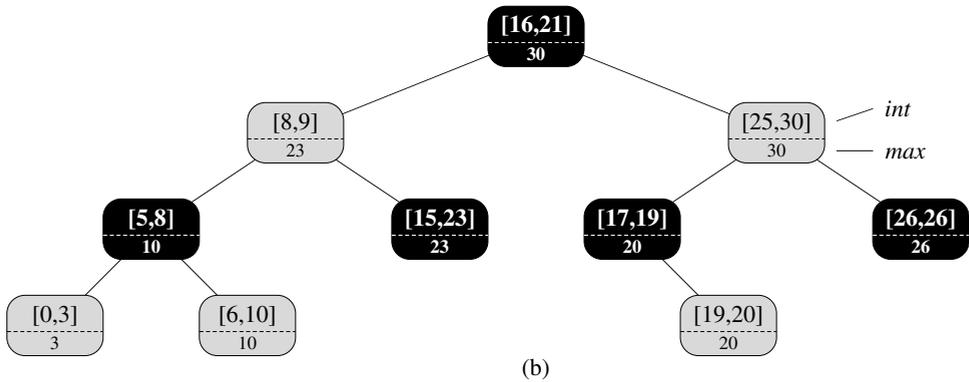
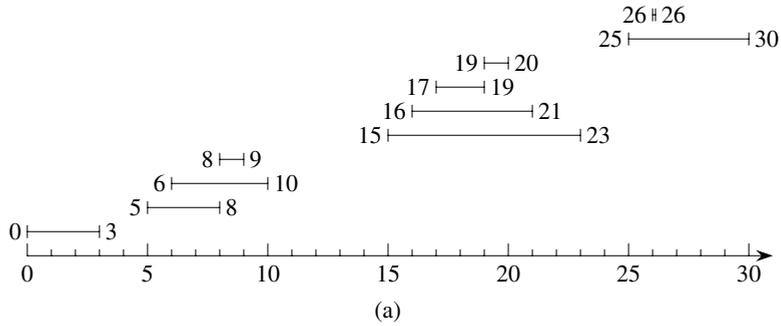


Figure 14.4 Un arbre d'intervalles. (a) Un ensemble de 10 intervalles, triés du bas vers le haut selon leur extrémité gauche. (b) L'arbre d'intervalles qui les représente. Un parcours préfixe de l'arbre recense les nœuds en les triant suivant leur extrémité gauche.

c) *Étape 3 : Mise à jour de l'information*

On doit vérifier que l'insertion et la suppression peuvent s'exécuter en $O(\lg n)$ sur un arbre d'intervalles à n nœuds. On peut déterminer $max[x]$ connaissant un intervalle $int[x]$ et les valeurs max des fils du nœud x :

$$max[x] = \max(fin[int[x]], max[gauche[x]], max[droite[x]]) .$$

Ainsi, d'après le théorème 14.1, l'insertion et la suppression ont un temps d'exécution en $O(\lg n)$. En fait, la mise à jour des champs max après une rotation peut s'effectuer en $O(1)$, comme le montrent les exercices 14.2.4 et 14.3.1.

d) *Étape 4 : Développer les nouvelles opérations*

La seule opération nouvelle nécessaire est INTERVALLE-RECHERCHER(T, i), qui trouve dans l'arbre T un nœud dont l'intervalle recoupe l'intervalle i . S'il n'existe aucun intervalle de ce type, on retourne un pointeur vers la sentinelle $nil[T]$.

```

RECHERCHER-INTERVALLE( $T, i$ )
1   $x \leftarrow \text{racine}[T]$ 
2  tant que  $x \neq \text{nil}[T]$  et  $i$  ne recoupe pas  $\text{int}[x]$ 
3      faire si  $\text{gauche}[x] \neq \text{nil}[T]$  et  $\max[\text{gauche}[x]] \geq \text{début}[i]$ 
4          alors  $x \leftarrow \text{gauche}[x]$ 
5          sinon  $x \leftarrow \text{droite}[x]$ 
6  retourner  $x$ 

```

La recherche d'un intervalle recoupant i commence avec x à la racine de l'arbre, et continue en descendant. Elle se termine soit quand un recoupement d'intervalles est trouvé, soit quand x pointe vers la sentinelle $\text{nil}[T]$. Comme chaque itération de la boucle principale prend un temps en $O(1)$, et que la hauteur d'un arbre rouge-noir à n nœuds est $O(\lg n)$, la procédure RECHERCHER-INTERVALLE prend $O(\lg n)$.

Avant de vérifier la validité de RECHERCHER-INTERVALLE, examinons son fonctionnement sur l'arbre d'intervalles de la figure 14.4. Supposons qu'on veuille trouver un intervalle qui recoupe l'intervalle $i = [22, 25]$. On commence avec x à la racine, qui contient $[16, 21]$, et ne recoupe pas i . Comme $\max[\text{gauche}[x]] = 23$ est plus grand que $\text{début}[i] = 22$, la boucle continue en prenant pour x la valeur du fils gauche de la racine, à savoir le nœud contenant l'intervalle $[8, 9]$, qui ne recoupe pas non plus i . Cette fois, $\max[\text{gauche}[x]] = 10$ est inférieur à $\text{début}[i] = 22$, et la boucle continue en prenant comme nouvel x le fils droit de x . L'intervalle $[15, 23]$ stocké dans ce nœud recoupe i , et c'est donc ce nœud qui est retourné par la procédure.

Comme exemple de recherche infructueuse, supposons qu'on souhaite trouver un intervalle qui recoupe $i = [11, 14]$ dans l'arbre d'intervalles de la figure 14.4. On commence une fois encore en prenant pour x la racine de l'arbre. Comme l'intervalle $[16, 21]$ situé à la racine ne recoupe pas i , et que $\max[\text{gauche}[x]] = 23$ est plus grand que $\text{début}[i] = 11$, on se dirige vers la gauche, sur le nœud contenant l'intervalle $[8, 9]$. (Notez qu'aucun intervalle du sous-arbre droit ne recoupe i , nous verrons pourquoi plus tard.) L'intervalle $[8, 9]$ ne recoupe pas i , et $\max[\text{gauche}[x]] = 10$ est inférieur à $\text{début}[i] = 11$, ce qui nous dirige vers la droite. (Notez qu'aucun intervalle du sous-arbre gauche ne recoupe i .) L'intervalle $[15, 23]$ ne recoupe pas i , et son fils gauche vaut $\text{nil}[T]$; on prend donc à droite, la boucle se termine, et la sentinelle $\text{nil}[T]$ est retournée.

Pour vérifier la validité de RECHERCHER-INTERVALLE, on doit comprendre qu'il suffit d'examiner un chemin unique partant de la racine. Le principe est qu'à un nœud x quelconque, si $\text{int}[x]$ ne recoupe pas i , la recherche continue toujours dans la bonne direction : nous sommes sûrs de trouver un recoupement s'il en existe un dans l'arbre. Le théorème suivant établit cette propriété plus précisément.

Théorème 14.2 *Chaque exécution de RECHERCHER-INTERVALLE(T, i) soit retourne un nœud dont l'intervalle recoupe i , soit retourne $\text{nil}[T]$ et l'arbre T ne contient aucun nœud dont l'intervalle recoupe i .*

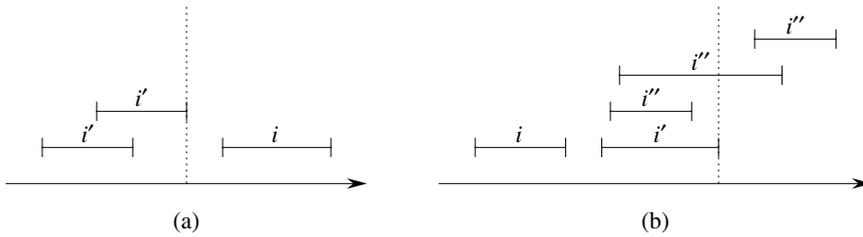


Figure 14.5 Intervalles dans la démonstration du théorème 14.2. La valeur de $\max[\text{gauche}[x]]$ est représentée dans chaque cas par une ligne pointillée. **(a)** La recherche se prolonge vers la droite. Aucun intervalle i' ne peut recouper i . **(b)** La recherche se prolonge vers la gauche. Le sous-arbre gauche de x contient un intervalle qui recoupe i (situation non représentée), ou il existe un intervalle i' dans le sous-arbre gauche de x tel que $\text{fin}[i'] = \max[\text{gauche}[x]]$. Comme i ne recoupe pas i' , il ne recoupe pas non plus un autre intervalle i'' quelconque du sous-arbre droit de x , puisque $\text{début}[i'] \leq \text{début}[i'']$.

Démonstration : La boucle **tant que** des lignes 2–5 se termine soit quand $x = \text{nil}[T]$, soit quand i recoupe $\text{int}[x]$. Dans le dernier cas, on sait avec certitude qu'il faut retourner x . Concentrons-nous donc sur le premier cas, où la boucle **tant que** se termine parce que $x = \text{nil}[T]$.

Nous emploierons l'invariant suivant pour la boucle **tant que** des lignes 2–5 :

Si l'arbre T contient un intervalle qui recoupe i , alors il existe un tel intervalle dans le sous-arbre issu de x .

Nous utiliserons cet invariant comme suit :

Initialisation : Avant la première itération, la ligne 1 prend pour x la racine de T , de sorte que l'invariant est vrai.

Conservation : Dans chaque itération de la boucle **tant que**, il y a exécution soit de la ligne 4, soit de la ligne 5. Nous allons montrer que l'invariant est conservé dans l'un ou l'autre cas.

Si c'est la ligne 5 qui est exécutée, compte tenu de la condition de branchement en ligne 3, on a $\text{gauche}[x] = \text{nil}[T]$, ou $\max[\text{gauche}[x]] < \text{début}[i]$. Si $\text{gauche}[x] = \text{nil}[T]$, le sous-arbre issu de $\text{gauche}[x]$ ne contient visiblement aucun intervalle qui recoupe i , et donc prendre pour x la valeur $\text{droite}[x]$ conserve l'invariant. Supposons donc que $\text{gauche}[x] \neq \text{nil}[T]$ et $\max[\text{gauche}[x]] < \text{début}[i]$. Comme le montre la figure 14.5(a), pour chaque intervalle i' du sous-arbre gauche de x , on a

$$\begin{aligned} \text{fin}[i'] &\leq \max[\text{gauche}[x]] \\ &< \text{début}[i] . \end{aligned}$$

D'après la trichotomie d'intervalle, i' et i ne se recoupent donc pas. Par conséquent, le sous-arbre gauche de x ne contient aucun intervalle qui recoupe i ; donc, prendre pour x la valeur $\text{droite}[x]$ conserve l'invariant.

Si, en revanche, c'est la ligne 4 qui est exécutée, alors on va montrer que la contraposée de l'invariant est vraie. C'est-à-dire que l'on va montrer que, s'il n'existe aucun intervalle recoupant i dans le sous-arbre issu de $\text{gauche}[x]$, alors nulle part dans l'arbre il n'y a d'intervalle qui recoupe i . Comme la ligne 4 est exécutée, compte

tenu de la condition de branchement en ligne 3, on a $\max[\text{gauche}[x]] \geq \text{début}[i]$. En outre, d'après la définition du champ \max , il doit obligatoirement exister un certain intervalle i' dans le sous-arbre gauche de x tel que

$$\begin{aligned} \text{fin}[i'] &= \max[\text{gauche}[x]] \\ &\geq \text{début}[i]. \end{aligned}$$

(La figure 14.5(b) illustre cette situation.) Comme i et i' ne se recoupent pas, et comme il n'est pas vrai que $\text{fin}[i'] < \text{début}[i]$, il s'ensuit, d'après la trichotomie d'intervalle, que $\text{fin}[i] < \text{début}[i']$. Comme les clés d'un arbre d'intervalles sont les débuts des intervalles, la propriété d'arbre de recherche implique que, pour tout intervalle i'' du sous-arbre droit de x' ,

$$\begin{aligned} \text{fin}[i] &< \text{début}[i'] \\ &\leq \text{début}[i'']. \end{aligned}$$

D'après la trichotomie d'intervalle, i et i'' ne se recoupent pas. On en conclut que, qu'il y ait ou non un intervalle du sous-arbre gauche de x' qui recoupe i , prendre pour x la valeur $\text{gauche}[x]$ conserve l'invariant.

Terminaison : Si la boucle se termine quand $x = \text{nil}[T]$, il n'y a pas d'intervalle qui recoupe i dans le sous-arbre issu de x . La contraposée de l'invariant de boucle implique que T ne contient aucun intervalle qui recoupe i . Il est donc correct de retourner $x = \text{nil}[T]$. \square

Par conséquent, la procédure RECHERCHER-INTERVALLE fonctionne correctement.

Exercices

14.3.1 Écrire un pseudo code pour ROTATION-GAUCHE agissant sur les nœuds d'un arbre d'intervalles et capable de mettre à jour les champs \max en $O(1)$.

14.3.2 Réécrire le code de RECHERCHER-INTERVALLE pour qu'il fonctionne correctement lorsque tous les intervalles sont supposés ouverts.

14.3.3 Décrire un algorithme efficace qui, étant donné un intervalle i , retourne un intervalle recoupant i qui a l'extrémité initiale minimale, ou qui retourne $\text{nil}[T]$ si un tel intervalle n'existe pas.

14.3.4 Étant donné un arbre d'intervalles T et un intervalle i , décrire comment tous les intervalles de T recoupant i peuvent être recensés en $O(\min(n, k \lg n))$, où k est le nombre d'intervalles présents dans la liste de sortie. (*Optionnel :* Trouver une solution qui ne modifie pas l'arbre.)

14.3.5 Suggérer des modifications aux procédures d'arbre d'intervalles permettant de supporter l'opération RECHERCHER-INTERVALLE-EXACT(T, i), qui retourne un pointeur sur un nœud x de l'arbre d'intervalles T tel que $début[int[x]] = début[i]$ et $fin[int[x]] = fin[i]$, ou qui retourne $nil[T]$ si T ne contient pas un tel nœud. Toutes les opérations, y compris RECHERCHER-INTERVALLE-EXACT, devront s'exécuter dans un temps en $O(\lg n)$ sur un arbre à n nœuds.

14.3.6 Montrer comment gérer un ensemble dynamique Q de nombres pouvant supporter l'opération DISTANCE-MIN, qui donne la longueur de la différence entre les deux nombres les plus proches dans Q . Par exemple, si $Q = \{1, 5, 9, 15, 18, 22\}$, alors DISTANCE-MIN(Q) retourne $18 - 15 = 3$, puisque 15 et 18 sont les nombres les plus proches dans Q . Rendre les opérations INSÉRER, SUPPRIMER, RECHERCHER et DISTANCE-MIN les plus efficaces possibles, et analyser leur temps d'exécution.

14.3.7 * Les bases de données VLSI représentent souvent un circuit intégré par une liste de rectangles. On admet que chaque rectangle est orienté de façon rectilinéaire (côtés parallèles aux axes x et y), de manière qu'on puisse représenter un rectangle par ses coordonnées x et y minimales et maximales. Donner un algorithme en $O(n \lg n)$ permettant de décider si un ensemble de rectangles représentés de cette manière contient deux rectangles qui se recourent. Votre algorithme n'a pas besoin de donner toutes les paires ayant une surface commune, mais il doit signaler qu'il y a recouvrement s'il existe un rectangle qui recouvre entièrement un autre, même si leurs frontières ne se coupent pas. (*Conseil* : Déplacer une ligne de « balayage » sur l'ensemble des rectangles.)

PROBLÈMES

14.1. Point de recouvrement maximal

On veut gérer le *point de recouvrement maximal* d'un ensemble d'intervalles, c'est-à-dire le point recouvert par le plus grand nombre d'intervalles de la base de données.

- a. Montrer qu'il y a toujours un point de recouvrement maximal qui est une extrémité de l'un des segments.
- b. Concevoir une structure de données qui permette de gérer efficacement les opérations INSÉRER-INTERVALLE, SUPPRIMER-INTERVALLE et TROUVER-PRM ; cette dernière procédure retourne un point de recouvrement maximal. (*conseil* : Gérer un arbre rouge-noir contenant toutes les extrémités. Associer la valeur +1 à chaque extrémité gauche et la valeur -1 à chaque extrémité droite. Augmenter chaque nœud de l'arbre en y ajoutant des données supplémentaires pour gérer le point de recouvrement maximal.)

14.2. Permutation de Josephus

Le *problème de Josephus* se définit de la façon suivante. On a n personnes rangées en cercle et un entier positif $m \leq n$. En commençant par une personne désignée, on fait le tour du cercle en éliminant une personne toutes les m . Après chaque élimination, on continue à compter avec le cercle qui reste. Le processus se poursuit jusqu'à ce que les n personnes aient été éliminées. L'ordre dans lequel les personnes sont éliminées du cercle définit la *permutation de Josephus* (n, m) des entiers $1, 2, \dots, n$. Par exemple, la permutation de Josephus $(7, 3)$ est $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

- a. Supposons que m soit une constante. Décrire un algorithme en $O(n)$ qui, étant donné un entier n , donne la permutation de Josephus (n, m) .
- b. On suppose que m n'est pas une constante. Décrire un algorithme en $O(n \lg n)$ qui, étant donnés les entiers n et m , donne la permutation de Josephus (n, m) .

NOTES

Dans leur livre, Preparata et Shamos [247] décrivent plusieurs types d'arbres d'intervalles que l'on trouve dans les manuels, citant notamment les travaux de H. Edelsbrunner (1980) et E. M. McCreight (1981). Le livre détaille un arbre d'intervalles pour lequel, étant donnée une base de données statique de n intervalles, les k intervalles qui recourent un certain intervalle donné sont énumérables en temps $O(k + \lg n)$.

PARTIE 4

TECHNIQUES AVANCÉES DE CONCEPTION ET D'ANALYSE

Cette partie aborde trois techniques importantes pour la conception et l'analyse d'algorithmes efficaces : la programmation dynamique (chapitre 15), les algorithmes gloutons (chapitre 16) et l'analyse amortie (chapitre 17). Les parties précédentes ont présenté d'autres techniques largement utilisées, comme l'approche diviser-pour-régner, la randomisation et les récurrences. Les nouvelles techniques sont un peu plus sophistiquées, mais elles sont essentielles pour s'attaquer efficacement à de nombreux problèmes informatiques. Les thèmes introduits dans cette partie réapparaîtront à plusieurs reprises dans la suite de ce livre.

La programmation dynamique s'applique le plus souvent à des problèmes d'optimisation dans lesquels on doit faire un ensemble de choix pour arriver à une solution optimale. À mesure que l'on fait des choix, on voit souvent surgir des sous-problèmes de la même forme. La programmation dynamique est efficace lorsqu'un sous-problème donné est susceptible d'être engendré par plus d'un ensemble partiel de choix ; le principe fondamental ici est de mémoriser la solution d'un tel sous-problème, pour le cas où il réapparaîtrait. Le chapitre 15 montrera comment cette idée simple peut parfois transformer des algorithmes à temps exponentiel en algorithmes à temps polynomial.

À l'instar des algorithmes de la programmation dynamique, les algorithmes gloutons s'appliquent en général à des problèmes d'optimisation dans lesquels il faut effectuer un ensemble de choix pour arriver à une solution optimale. Le principe d'un algorithme glouton est de faire chaque choix d'une manière qui soit localement

optimale. Un exemple simple en est donné par l'opération consistant à rendre la monnaie : pour minimiser le nombre de pièces utilisées pour rendre la monnaie pour un montant donné, il suffit de choisir de manière répétée la pièce la plus forte inférieure ou égale au reste dû. Il existe de nombreux problèmes de cette sorte pour lesquels une approche gloutonne fournit une solution optimale bien plus rapidement que ne le ferait la programmation dynamique. Toutefois, il n'est pas toujours facile de prédire si une approche gloutonne serait efficace. Le chapitre 16 donne un aperçu de la théorie des matroïdes qui facilite souvent ce genre d'estimation.

L'analyse amortie permet d'analyser un algorithme qui effectue une séquence d'opérations similaires. Au lieu de borner le coût de la séquence en bornant le coût réel de chaque opération séparément, on utilise une analyse amortie pour borner le coût réel de la séquence globale. L'idée sous-jacente ici est que toutes les opérations de la séquence ne vont pas s'effectuer systématiquement dans le contexte du cas le plus défavorable. Certaines opérations seront coûteuses, mais d'autres non. Par ailleurs, l'analyse amortie n'est pas seulement un outil d'analyse ; c'est aussi une technique de conception d'algorithmes, car conception et analyse du temps d'exécution sont des concepts qui sont souvent très imbriqués. Le chapitre 17 présentera trois façons de faire l'analyse amortie d'un algorithme.

Chapitre 15

Programmation dynamique

La programmation dynamique, comme la méthode diviser-pour-régner, résout des problèmes en combinant des solutions de sous-problèmes. (« Programmation », dans ce contexte, fait référence à une méthode tabulaire et non à l'écriture de code informatique.). Comme nous l'avons vu au chapitre 2, les algorithmes diviser-pour-régner partitionnent le problème en sous-problèmes indépendants qu'ils résolvent récursivement, puis combinent leurs solutions pour résoudre le problème initial. La programmation, quant à elle, peut s'appliquer même lorsque les sous-problèmes ne sont pas indépendants, c'est-à-dire lorsque des sous-problèmes ont des sous-sous-problèmes communs. Dans ce cas, un algorithme diviser-pour-régner fait plus de travail que nécessaire, en résolvant plusieurs fois le sous-sous-problème commun. Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcul de la solution chaque fois que le sous-sous-problème est rencontré.

La programmation dynamique est, en général, appliquée aux *problèmes d'optimisation*. Dans ce type de problèmes, il peut y avoir de nombreuses solutions possibles. Chaque solution a une valeur, et on souhaite trouver une solution ayant la valeur optimale (minimale ou maximale). Une telle solution est *une* solution optimale au problème, et non pas *la* solution optimale, puisqu'il peut y avoir plusieurs solutions qui donnent la valeur optimale.

Le développement d'un algorithme de programmation dynamique peut être découpé en quatre étapes.

- 1) Caractériser la structure d'une solution optimale.

- 2) Définir récursivement la valeur d'une solution optimale.
- 3) Calculer la valeur d'une solution optimale de manière ascendante (bottom-up).
- 4) Construire une solution optimale à partir des informations calculées.

Les étapes 1–3 forment la base d'une résolution de problème à la mode de la programmation dynamique. On peut omettre l'étape 4 si l'on n'a besoin que de la valeur d'une solution optimale. Lorsqu'on effectue l'étape 4, on gère parfois des informations supplémentaires pendant le calcul de l'étape 3 pour faciliter la construction d'une solution optimale.

Les sections suivantes utilisent la programmation dynamique pour résoudre certains problèmes d'optimisation. La section 15.1 étudie un problème concernant l'ordonnement de deux chaînes de montage d'automobiles où, après chaque poste, l'auto en cours de fabrication peut rester sur la même chaîne ou passer sur l'autre chaîne. La section 15.2 montre comment multiplier une suite de matrices avec le moins possible de multiplications scalaires. Partant de ces exemples de programmation dynamique, la section 15.3 étudie deux grandes caractéristiques que doit posséder un problème pour que la programmation dynamique soit une technique de résolution viable. La section 15.4 montre ensuite comment trouver la plus longue sous-séquence commune de deux séquences. Enfin, la section 15.5 emploie la programmation dynamique pour construire des arbres binaires de recherche qui sont optimaux, étant donnée une distribution connue des clés à chercher.

15.1 ORDONNANCEMENT DE CHAÎNES DE MONTAGE

Notre premier exemple de programmation dynamique traite d'un problème de fabrication. La firme Facel-Véga produit des automobiles dans un atelier qui a deux chaînes de montage (voir figure 15.1). Un châssis arrive sur chaque chaîne, puis passe par un certain nombre de postes où on lui ajoute des pièces ; une fois terminée, l'auto sort par l'autre extrémité de la chaîne. Chaque chaîne de montage a n stations, numérotées $j = 1, 2, \dots, n$. On représente le j ème poste de la chaîne i (avec i égale 1 ou 2) par $S_{i,j}$. Le j ème poste de la chaîne 1 ($S_{1,j}$) fait le même travail que le j ème poste de la chaîne 2 ($S_{2,j}$). Les postes ont été installés à des époques différentes et avec des technologies différentes ; ainsi, le temps de montage varie d'un poste à l'autre, même quand il s'agit de postes fonctionnement identiques mais situés sur des chaînes différentes. Le temps de montage au poste $S_{i,j}$ est $a_{i,j}$. Comme le montre la figure 15.1, un châssis arrive au poste 1 de l'une des chaînes, puis passe de poste en poste. On a aussi un temps d'arrivée e_i pour le châssis qui entre sur la chaîne i , et un temps de sortie x_i pour l'auto achevée qui sort de la chaîne i .

Normalement, une fois qu'un châssis arrive sur une chaîne de montage, il ne circule que sur cette chaîne. Le temps de passage d'un poste à l'autre sur une même chaîne est négligeable. En cas d'urgence, toutefois, il se peut que l'on veuille accélérer le délai de fabrication d'une automobile. En pareil cas, le châssis transite toujours

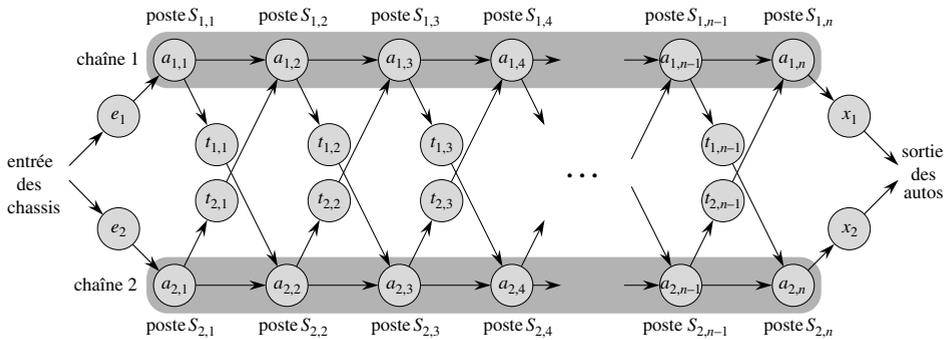


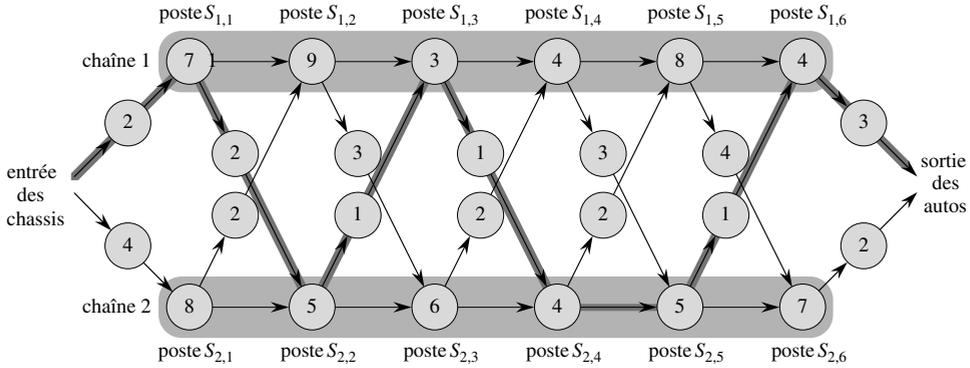
Figure 15.1 Problème de fabrication pour déterminer le chemin optimal dans une usine. Il y a deux chaînes de montage, ayant chacune n postes ; le j ème poste de la chaîne i est noté $S_{i,j}$ et le temps de montage à ce poste est $a_{i,j}$. Un châssis entre dans l'atelier, puis va sur la chaîne i (avec $i = 1$ ou 2) en mettant un temps e_i . Après être passé par le j ème poste d'une chaîne, le châssis va sur le $(j+1)$ ème poste de l'une ou l'autre chaîne. Il n'y a pas de coût de transfert si l'auto reste sur la même chaîne, mais il faut un temps $t_{i,j}$ pour passer sur l'autre chaîne après le poste $S_{i,j}$. Après avoir quitté le n ème poste d'une chaîne, l'auto achevée met un temps x_i pour sortir de l'atelier. Le problème consiste à déterminer quels sont les postes à sélectionner sur la chaîne 1 et sur la chaîne 2 pour minimiser le délai de transit d'une auto à travers l'atelier.

par les n postes dans l'ordre, mais le chef d'atelier peut faire passer une auto partiellement construite d'une chaîne à l'autre, et ce après chaque poste. Le temps de transfert d'un châssis depuis la chaîne i et après le poste $S_{i,j}$ est $t_{i,j}$, avec $i = 1, 2$ et $j = 1, 2, \dots, n - 1$ (car, après le n ème poste, c'est fini). Le problème consiste à déterminer quels sont les postes à sélectionner sur la chaîne 1 et sur la chaîne 2 pour minimiser le délai de transit d'une auto à travers l'atelier. Sur l'exemple de la figure 15.2(a), le délai optimal est obtenu *via* sélection des postes 1, 3 et 6 de la chaîne 1 et des postes 2, 4 et 5 de la chaîne 2.

La solution évidente et « primaire », pour minimiser le délai de circulation à travers l'atelier, est irréaliste quand il y a beaucoup de postes. Connaissant la liste des postes à utiliser sur la chaîne 1 et des postes à utiliser sur la chaîne 2, il est facile de calculer en temps $\Theta(n)$ le délai de transit d'un châssis à travers l'atelier. Malheureusement, il y a 2^n façons possibles de choisir les postes ; on peut le voir en considérant l'ensemble des postes utilisés sur la chaîne 1 comme un sous-ensemble de $\{1, 2, \dots, n\}$ et en remarquant qu'il y a 2^n tels sous-ensembles. Par conséquent, déterminer le chemin le plus rapide en énumérant tous les chemins possibles puis en calculant la durée de chacun, c'est une solution qui prend un temps $\Omega(2^n)$, ce qui est irréaliste quand n est grand.

a) Étape 1 : structure du chemin optimal à travers l'atelier

La première étape du paradigme de la programmation dynamique est de caractériser la structure d'une solution optimale. Pour le problème de l'ordonnement de chaîne de montage, voici comment on peut procéder à cette étape. Considérons le chemin



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Figure 15.2 (a) Une instance du problème de la chaîne de montage, avec les coûts e_i , $a_{i,j}$, $t_{i,j}$ et x_i affichés. Le chemin sur fond gris foncé indique le chemin le plus rapide à travers l'atelier. (b) Les valeurs de $f_1[j]$, f^* , $l_1[j]$ et l^* pour l'instance de la partie (a).

optimal pour aller du point de départ au poste $S_{1,j}$. Si $j = 1$, il n'y a qu'un seul chemin possible et il est donc facile de déterminer le temps qu'il faut pour arriver au poste $S_{1,j}$. Pour $j = 2, 3, \dots, n$, en revanche, il y a deux possibilités : le châssis peut aller du poste $S_{1,j-1}$ au poste $S_{1,j}$ directement, le délai de passage du poste $j - 1$ au poste j de la même chaîne étant négligeable. Mais le châssis peut aussi aller du poste $S_{2,j-1}$ au poste $S_{1,j}$, le délai de transfert étant alors $t_{2,j-1}$. Nous traiterons séparément ces deux cas de figure, bien qu'ils aient beaucoup de points communs comme nous le verrons.

Primo, supposons que le chemin optimal vers le poste $S_{1,j}$ passe par le poste $S_{1,j-1}$. La remarque fondamentale est que le châssis a forcément pris un chemin optimal pour aller du point de départ au poste $S_{1,j-1}$. Pourquoi ? S'il existait un chemin plus rapide pour aller au poste $S_{1,j-1}$, on pourrait utiliser ce chemin plus rapide pour obtenir un chemin plus rapide vers le poste $S_{1,j}$; d'où une contradiction.

De même, supposons maintenant que le chemin optimal vers le poste $S_{1,j}$ passe par le poste $S_{2,j-1}$. Le châssis a forcément pris un chemin optimal du point de départ au poste $S_{2,j-1}$. Le raisonnement est le même : s'il existait un chemin plus rapide menant au poste $S_{2,j-1}$, on utiliserait ce chemin plus rapide pour obtenir un chemin plus rapide vers le poste $S_{1,j}$; d'où une contradiction.

Plus généralement, on peut dire que, pour l'ordonnement de chaîne de montage, une solution optimale à un problème (trouver le chemin optimal menant au

poste $S_{i,j}$) contient en elle une solution optimale pour des sous-problèmes (trouver le chemin optimal menant à $S_{1,j-1}$ ou $S_{2,j-1}$). On parle ici de propriété de **sous-structure optimale**, et c'est l'un des grands critères de l'applicabilité de la programmation dynamique, comme nous le verrons à la section 15.3.

On emploie la sous-structure optimale pour montrer que l'on peut construire une solution optimale pour un problème à partir de solutions optimales pour des sous-problèmes. Pour l'ordonnement de chaîne de montage, on raisonne comme suit. Si l'on examine un chemin optimal menant au poste $S_{1,j}$, il doit passer par le poste $j-1$ de l'une des chaînes 1 et 2. Donc, le chemin optimal passant par le poste $S_{1,j}$ est soit

- le chemin optimal passant par le poste $S_{1,j-1}$ puis allant directement au poste $S_{1,j}$, soit
- le chemin optimal passant par le poste $S_{2,j-1}$, sautant de la chaîne 2 à la chaîne 1, puis allant au poste $S_{1,j}$.

Un raisonnement symétrique dit que le chemin optimal passant par le poste $S_{2,j}$ est soit

- le chemin optimal passant par le poste $S_{2,j-1}$ puis allant directement au poste $S_{2,j}$, soit
- le chemin optimal passant par le poste $S_{1,j-1}$, sautant de la chaîne 1 à la chaîne 2, puis allant au poste $S_{2,j}$.

Pour résoudre le problème du calcul du chemin optimal passant par le poste j de l'une ou l'autre chaîne, on résout les sous-problèmes consistant à calculer les chemins optimaux passant par le poste $j-1$ des deux chaînes de montage.

Par conséquent, on peut construire une solution optimale d'une instance de l'ordonnement de chaîne de montage partir de la construction de solutions optimales pour des sous-problèmes.

b) Étape 2 : une solution récursive

La deuxième étape du paradigme de la programmation dynamique consiste à définir la valeur d'une solution optimale de manière récursive à partir de solutions optimales de sous-problèmes. Pour le problème de l'ordonnement de chaîne de montage, on choisit comme sous-problèmes les problèmes consistant à trouver le chemin optimal passant par le poste j des deux chaînes, pour $j = 1, 2, \dots, n$. Soit $f_i[j]$ le délai le plus court possible avec lequel le châssis va du point de départ au poste $S_{i,j}$.

Notre objectif ultime est de déterminer le délai le plus court par lequel un châssis traverse tout l'atelier, délai que nous noterons f^* . Le châssis doit aller au poste n de l'une ou l'autre des chaînes 1 et 2, et de là aller vers la sortie de l'atelier. Comme le plus rapide de ces chemins est le chemin optimal à travers tout l'atelier, on a

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2). \quad (15.1)$$

Il est facile aussi de raisonner sur $f_1[1]$ et $f_2[1]$. Pour passer par le poste 1 d'une des deux chaînes, un châssis va tout simplement vers ce poste directement. Donc,

$$f_1[1] = e_1 + a_{1,1} , \quad (15.2)$$

$$f_2[1] = e_2 + a_{2,1} . \quad (15.3)$$

Voyons maintenant comment calculer $f_i[j]$ pour $j = 2, 3, \dots, n$ (et $i = 1, 2$). En nous focalisant sur $f_1[j]$, rappelons-nous que le chemin optimal passant par le poste $S_{1,j}$ est soit le chemin optimal passant par le poste $S_{1,j-1}$ suivi du passage direct au poste $S_{1,j}$, soit le chemin optimal passant par le poste $S_{2,j-1}$ suivi d'un transfert de la chaîne 2 à la chaîne 1 et suivi enfin du passage au poste $S_{1,j}$. Dans le premier cas, on a $f_1[j] = f_1[j-1] + a_{1,j}$; dans le dernier cas, on a $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$. Par conséquent,

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) \quad (15.4)$$

pour $j = 2, 3, \dots, n$. De manière symétrique, on a

$$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) \quad (15.5)$$

pour $j = 2, 3, \dots, n$. En combinant les équations (15.2)–(15.5), on obtient les équations récursives

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{si } j = 1 , \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{si } j \geq 2 \end{cases} \quad (15.6)$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{si } j = 1 , \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{si } j \geq 2 \end{cases} \quad (15.7)$$

La figure 15.2(b) montre les valeurs $f_i[j]$ pour l'exemple de la partie (a), telles que calculées par les équations (15.6) et (15.7), avec en plus la valeur de f^* .

Les valeurs $f_i[j]$ donnent les valeurs de solutions optimales pour des sous-problèmes. Pour nous aider à comprendre comment on construit une solution optimale, définissons $l_i[j]$ comme étant le numéro de la chaîne de montage, 1 ou 2, dont le poste $j-1$ est utilisé par un chemin optimal passant par le poste $S_{i,j}$. Ici, $i = 1, 2$ et $j = 2, 3, \dots, n$. (On évite de définir $l_i[1]$, car aucun poste ne vient avant le poste 1 sur l'une ou l'autre chaîne.) On définit aussi l^* comme étant la chaîne dont le poste n est utilisé par un chemin optimal traversant tout l'atelier. Les valeurs $l_i[j]$ facilitent le tracé d'un chemin optimal. En utilisant les valeurs de l^* et de $l_i[j]$ montrées à la figure 15.2(b), voici comment on tracerait un chemin optimal à travers l'atelier de la partie (a). On part avec $l^* = 1$, et on utilise le poste $S_{1,6}$. On regarde ensuite $l_1[6]$ qui vaut 2, et donc on utilise le poste $S_{2,5}$. En continuant, on regarde $l_2[5] = 2$ (utiliser poste $S_{2,4}$), $l_2[4] = 1$ (poste $S_{1,3}$), $l_1[3] = 2$ (poste $S_{2,2}$) et $l_2[2] = 1$ (poste $S_{1,1}$).

c) Étape 3 : calcul des temps optimaux

À ce stade, ce serait chose simple que d'écrire un algorithme récursif, à partir de l'équation (15.1) et des récurrences (15.6) et (15.7), pour calculer le chemin optimal à travers l'atelier. Il y a un hic avec un tel algorithme récursif : son temps d'exécution est exponentiel en n . Pour comprendre pourquoi, notons $r_i(j)$ le nombre de références faites à $f_i[j]$ dans un algorithme récursif. Partant de l'équation (15.1), on a

$$r_1(n) = r_2(n) = 1 . \quad (15.8)$$

Partant des récurrences (15.6) et (15.7), on a

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1) \quad (15.9)$$

pour $j = 1, 2, \dots, n-1$. Ainsi que l'exercice 15.1.2 vous demandera de le montrer, $r_i(j) = 2^{n-j}$. Donc, $f_1[1]$ à elle seule est référencée 2^{n-1} fois ! Comme l'exercice 15.1.3 vous demandera de le montrer, le nombre total de références à toutes les valeurs $f_i[j]$ est $\Theta(2^n)$.

On peut faire beaucoup mieux, en calculant les valeurs $f_i[j]$ dans un ordre différent de la façon récursive. Notez que, pour $j \geq 2$, chaque valeur de $f_i[j]$ dépend uniquement des valeurs de $f_1[j-1]$ et de $f_2[j-1]$. En calculant les valeurs $f_i[j]$ par ordre croissant de numéros j de poste (de la gauche vers la droite, sur la figure 15.2(b)), on peut calculer le chemin optimal à travers l'atelier, et le temps qu'il prend, en un temps $\Theta(n)$. La procédure PLUS-RAPIDE-CHEMIN prend en entrée les valeurs $a_{i,j}$, $t_{i,j}$, e_i et x_i , plus n , nombre de postes de chaque chaîne de montage.

PLUS-RAPIDE-CHEMIN(a, t, e, x, n)

```

1   $f_1[1] \leftarrow e_1 + a_{1,1}$ 
2   $f_2[1] \leftarrow e_2 + a_{2,1}$ 
3  pour  $j \leftarrow 2$  à  $n$ 
4      faire si  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
5          alors  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$ 
6               $l_1[j] \leftarrow 1$ 
7          sinon  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
8               $l_1[j] \leftarrow 2$ 
9      si  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
10         alors  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$ 
11              $l_2[j] \leftarrow 2$ 
12         sinon  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
13              $l_2[j] \leftarrow 1$ 
14 si  $f_1[n] + x_1 \leq f_2[n] + x_2$ 
15     alors  $f^* = f_1[n] + x_1$ 
16          $l^* = 1$ 
17     sinon  $f^* = f_2[n] + x_2$ 
18          $l^* = 2$ 

```

PLUS-RAPIDE-CHEMIN fonctionne de la manière suivante. Les lignes 1–2 calculent $f_1[1]$ et $f_2[1]$ en utilisant les équations (15.2) et (15.3). Ensuite, la boucle **pour** des lignes 3–13 calcule $f_i[j]$ et $l_i[j]$ pour $i = 1, 2$ et $j = 2, 3, \dots, n$. Les lignes 4–8 calculent $f_1[j]$ et $l_1[j]$ en utilisant l'équation (15.4), et les lignes 9–13 calculent $f_2[j]$ et $l_2[j]$ en utilisant l'équation (15.5). Enfin, les lignes 14–18 calculent f^* et l^* en utilisant l'équation (15.1). Comme les lignes 1–2 et 14–18 prennent un temps constant et que chacune des $n - 1$ itérations de la boucle **pour** des lignes 3–13 prend un temps constant, l'ensemble de la procédure prend un temps $\Theta(n)$.

Une façon de voir le calcul des valeurs de $f_i[j]$ et $l_i[j]$ est de considérer que l'on saisit des données dans un tableau. Si l'on prend l'exemple de la figure 15.2(b), on remplit des tableaux contenant les valeurs $f_i[j]$ et $l_i[j]$ de gauche à droite (et de haut en bas dans chaque colonne). Pour remplir une case $f_i[j]$, on a besoin des valeurs de $f_1[j - 1]$ et de $f_2[j - 1]$; ensuite, sachant que l'on a déjà calculé et mémorisé ces valeurs, on les détermine rien qu'en consultant la table.

d) Étape 4 : construction du chemin optimal à travers l'atelier

Ayant calculé les valeurs de $f_i[j]$, f^* , $l_i[j]$ et l^* , on doit ensuite construire la séquence des postes utilisés par le chemin optimal traversant l'atelier. On a vu comment faire sur l'exemple de la figure 15.2.

La procédure suivante affiche les postes utilisés, par ordre décroissant de numéro de poste. L'exercice 15.1.1 vous demandera de modifier cette procédure pour qu'elle affiche les postes par ordre croissant de numéro.

```
AFFICHER-POSTES( $l, n$ )
```

```
1   $i \leftarrow l^*$ 
2  afficher « chaîne »  $i$  « , poste »  $n$ 
3  pour  $j \leftarrow n$  jusqu'à 2
4      faire  $i \leftarrow l_i[j]$ 
5          afficher « chaîne »  $i$  « , poste »  $j - 1$ 
```

Dans l'exemple de la figure 15.2, AFFICHER-POSTES sortirait le résultat que voici

```
chaîne 1, poste 6
chaîne 2, poste 5
chaîne 2, poste 4
chaîne 1, poste 3
chaîne 2, poste 2
chaîne 1, poste 1
```

Exercices

15.1.1 Montrer comment modifier AFFICHER-POSTES pour qu'elle affiche les postes par ordre de numéro croissant. (*Conseil* : Utiliser la récursivité.)

15.1.2 Utiliser les équations (15.8) et (15.9), ainsi que la méthode de substitution, pour montrer que $r_i(j)$, nombre de références faites à $f_i[j]$ dans un algorithme récursif, est égal à 2^{n-j} .

15.1.3 En utilisant le résultat de l'exercice 15.1.2, montrer que le nombre total de références à toutes les valeurs $f_i[j]$, soit $\sum_{i=1}^2 \sum_{j=1}^n r_i(j)$, vaut exactement $2^{n+1} - 2$.

15.1.4 Pris ensemble, les tableaux contenant les valeurs $f_i[j]$ et $l_i[j]$ renferment en tout $4n - 2$ éléments. Montrer comment ramener ce nombre à $2n + 2$ éléments, tout en conservant la possibilité de calculer f^* et d'afficher tous les postes d'un chemin optimal traversant l'atelier.

15.1.5 Le professeur Nulhardt pense qu'il pourrait y avoir des valeurs e_i , $a_{i,j}$ et $t_{i,j}$ pour lesquelles PLUS-RAPIDE-CHEMIN produirait des valeurs $l_i[j]$ telles que $l_1[j] = 2$ et $l_2[j] = 1$ pour un certain numéro de poste j . En supposant que tous les coûts de transfert $t_{i,j}$ soient positifs, montrer que le professeur se trompe.

15.2 MULTIPLICATIONS MATRICIELLES ENCHAÎNÉES

Notre exemple suivant de programmation dynamique est un algorithme qui résout le problème des multiplications matricielles enchaînées. On suppose qu'on a une chaîne $\langle A_1, A_2, \dots, A_n \rangle$ de n matrices à multiplier, et qu'on souhaite calculer le produit

$$A_1 A_2 \cdots A_n . \quad (15.10)$$

Il est possible d'évaluer l'expression (15.10) en utilisant, comme sous-programme, l'algorithme classique pour multiplier les paires de matrices, après avoir placé des parenthèses pour supprimer toute ambiguïté sur l'ordre de multiplication des matrices. Un produit de matrices *entièrement parenthésé* est soit une matrice unique, soit le produit de deux produits matriciels entièrement parenthésés. La multiplication des matrices est associative, et tous les parenthésages aboutissent donc à une même valeur du produit. Par exemple, si la chaîne de matrices est $\langle A_1, A_2, A_3, A_4 \rangle$, le produit $A_1 A_2 A_3 A_4$ peut être entièrement parenthésé de cinq façons distinctes :

$$\begin{aligned} & (A_1(A_2(A_3A_4))) , \\ & (A_1((A_2A_3)A_4)) , \\ & ((A_1A_2)(A_3A_4)) , \\ & ((A_1(A_2A_3))A_4) , \\ & (((A_1A_2)A_3)A_4) . \end{aligned}$$

La manière dont une suite de matrices est parenthésée peut avoir un impact crucial sur le coût d'évaluation du produit. Commençons par considérer le coût de la multiplication de deux matrices. L'algorithme standard est donné dans le pseudo code suivant. Les attributs *lignes* et *colonnes* représentent le nombre de lignes et de colonnes d'une matrice.

```

MULTIPLIER-MATRICES( $A, B$ )
1  si colonnes[ $A$ ]  $\neq$  lignes[ $B$ ]
2    alors erreur « dimensions incompatibles »
3    sinon pour  $i \leftarrow 1$  à lignes[ $A$ ]
4      faire pour  $j \leftarrow 1$  à colonnes[ $B$ ]
5        faire  $C[i, j] \leftarrow 0$ 
6          pour  $k \leftarrow 1$  à colonnes[ $A$ ]
7            faire  $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$ 
8    retourner  $C$ 

```

On ne peut multiplier deux matrices A et B que si elles sont *compatibles* : le nombre de colonnes de A est égal au nombre de lignes de B . Si A est une matrice $p \times q$ et B une matrice $q \times r$, la matrice résultante C est une matrice $p \times r$. Le temps de calcul de C est dominé par le nombre de multiplications scalaires de la ligne 7, qui vaut pqr . Dans ce qui suit, nous exprimerons les temps d'exécution en fonction du nombre de multiplications scalaires.

Pour illustrer les différents coûts induits par différents parenthésages d'un produit de matrices, on considère le problème d'une suite $\langle A_1, A_2, A_3 \rangle$ de trois matrices. On suppose que les dimensions des matrices sont respectivement 10×100 , 100×5 et 5×50 . Si l'on multiplie selon le parenthésage $((A_1A_2)A_3)$, on effectue $10 \cdot 100 \cdot 5 = 5000$ multiplications scalaires pour calculer la matrice produit 10×5 A_1A_2 , plus $10 \cdot 5 \cdot 50 = 2500$ multiplications scalaires pour multiplier cette matrice par A_3 , pour un total de 7 500 multiplications scalaires. Si on multiplie selon le parenthésage $(A_1(A_2A_3))$, on effectue $100 \cdot 5 \cdot 50 = 25\,000$ multiplications scalaires pour calculer la matrice produit 100×50 A_2A_3 , plus $10 \cdot 100 \cdot 50 = 50\,000$ multiplications pour multiplier A_1 par cette matrice, pour un total de 75 000 multiplications scalaires. Le calcul du produit selon le premier parenthésage est donc 10 fois plus rapide.

Le *problème des multiplications matricielles enchaînées* peut être énoncé comme suit : étant donnée une chaîne $\langle A_1, A_2, \dots, A_n \rangle$ de n matrices où, pour $i = 1, 2, \dots, n$, la matrice A_i a la dimension $p_{i-1} \times p_i$, parenthéser entièrement le produit $A_1A_2 \cdots A_n$ de façon à minimiser le nombre de multiplications scalaires.

Notez que, dans le problème des multiplications matricielles enchaînées, on ne multiplie pas vraiment les matrices. On cherche simplement un ordre de multiplication qui minimise le coût. Généralement, le temps passé à déterminer cet ordre optimal est plus que compensé par le gain de temps que l'on obtiendra quand on fera les multiplications matricielles proprement dites (par exemple, quand on fera 7 500 multiplications scalaires au lieu de 75 000).

a) Comptabilisation du nombre de parenthésages

Avant de résoudre le problème des multiplications matricielles enchaînées via la programmation dynamique, vérifions que passer en revue tous les parenthésages possibles ne donne pas un algorithme efficace. Soit $P(n)$ le nombre de parenthésages

possibles d'une séquence de n matrices. Quand $n = 1$, il n'y a qu'une seule matrice et donc une seule façon de parenthéser entièrement le produit. Quand $n \geq 2$, un produit matriciel entièrement parenthésé est le produit de deux sous-produits matriciels entièrement parenthésés, et la démarcation entre les deux sous-produits peut intervenir entre les k ème et $(k + 1)$ ème matrices, pour tout $k = 1, 2, \dots, n - 1$. On obtient donc la récurrence

$$P(n) = \begin{cases} 1 & \text{si } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2. \end{cases} \quad (15.11)$$

Le problème 12.4 vous demandait de montrer que la solution d'une récurrence similaire est la suite des **nombres de Catalan** qui croît en $\Omega(4^n/n^{3/2})$. Un exercice plus simple (voir exercice 15.2.3) consiste à montrer que la solution de la récurrence (15.11) est $\Omega(2^n)$. Le nombre de solutions est donc exponentiel en n , et la méthode primitive d'examen exhaustif est, de ce fait, une piètre stratégie pour déterminer le parenthésage optimal de produits matriciels enchaînés.

b) Structure d'un parenthésage optimal

La première étape de la programmation dynamique consiste à trouver la sous-structure optimale, puis à s'en servir pour construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. Pour le problème des multiplications matricielles enchaînées, on peut aborder cette étape de la manière suivante. Par commodité, nous adopterons la notation $A_{i..j}$, avec $i \leq j$, pour la matrice résultant de l'évaluation du produit $A_i A_{i+1} \cdots A_j$. Notez que, si le problème est non trivial, c'est-à-dire si $i < j$, alors tout parenthésage optimal du produit $A_i A_{i+1} \cdots A_j$ sépare le produit entre A_k et A_{k+1} pour un certain k de l'intervalle $i \leq k < j$. Autrement dit, pour une certaine valeur de k , on commence par calculer les matrices $A_{i..k}$ et $A_{k+1..j}$, puis on les multiplie ensemble pour obtenir le résultat final $A_{i..j}$. Le coût de ce parenthésage optimal est donc le coût du calcul de la matrice $A_{i..k}$, plus celui du calcul de $A_{k+1..j}$, plus celui de la multiplication de ces deux matrices.

La sous-structure optimale de ce problème est la suivante. Supposons qu'un parenthésage optimal de $A_i A_{i+1} \cdots A_j$ fractionne le produit entre A_k et A_{k+1} . Alors, le parenthésage de la sous-chaîne « préfixe » $A_i A_{i+1} \cdots A_k$ à l'intérieur de ce parenthésage optimal de $A_i A_{i+1} \cdots A_j$ est forcément un parenthésage optimal de $A_i A_{i+1} \cdots A_k$. Pourquoi ? S'il existait un parenthésage plus économique de $A_i A_{i+1} \cdots A_k$, substituer ce parenthésage dans le parenthésage optimal de $A_i A_{i+1} \cdots A_j$ produirait un autre parenthésage de $A_i A_{i+1} \cdots A_j$ dont le coût serait inférieur à l'optimum : on arrive à une contradiction. On peut faire la même observation pour le parenthésage de la sous-chaîne $A_{k+1} A_{k+2} \cdots A_j$ à l'intérieur du parenthésage optimal de $A_i A_{i+1} \cdots A_j$: c'est forcément un parenthésage optimal de $A_{k+1} A_{k+2} \cdots A_j$.

Utilisons maintenant notre sous-structure optimale pour montrer que nous pouvons construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. Nous avons vu que toute solution d'une instance non triviale du problème des multiplications matricielles enchaînées nous oblige à fractionner le produit, et que toute solution optimale contient en elle des solutions optimales d'instances de sous-problème. Par conséquent, on peut construire une solution optimale d'une instance du problème des multiplications matricielles enchaînées en fractionnant le problème en deux sous-problèmes (parenthésage optimal de $A_i A_{i+1} \cdots A_k$ et de $A_{k+1} A_{k+2} \cdots A_j$), en cherchant des solutions optimales d'instances de sous-problème, puis en combinant ces solutions optimales de sous-problème. Nous devons faire en sorte, quand nous recherchons l'endroit où fractionner le produit, de considérer tous les emplacements possibles ; ainsi, nous serons certains d'avoir choisi l'emplacement optimal.

c) Une solution récursive

La deuxième étape de la programmation dynamique consiste à définir récursivement le coût d'une solution optimale en fonction des solutions optimales de sous-problèmes. Pour le problème des multiplications matricielles enchaînées, on prend comme sous-problèmes les problèmes consistant à déterminer le coût minimal d'un parenthésage de $A_i A_{i+1} \cdots A_j$ pour $1 \leq i \leq j \leq n$. Soit $m[i, j]$ le nombre minimal de multiplications scalaires nécessaires pour le calcul de la matrice $A_{i..j}$; pour le problème entier, le coût de la méthode la plus économique pour calculer $A_{1..n}$ sera donc $m[1, n]$.

On peut définir $m[i, j]$ récursivement de la manière suivante. Si $i = j$, le problème est trivial ; la chaîne est constituée d'une seule matrice $A_{i..i} = A_i$, et aucune multiplication n'est nécessaire pour calculer le produit. Donc, $m[i, i] = 0$ pour $i = 1, 2, \dots, n$. Pour calculer $m[i, j]$ quand $i < j$, on exploite la structure d'une solution optimale définie à l'étape 1. Supposons que le parenthésage optimal sépare le produit $A_i A_{i+1} \cdots A_j$ entre A_k et A_{k+1} , avec $i \leq k < j$. Alors, $m[i, j]$ est égal au coût minimal du calcul des sous-produits $A_{i..k}$ et $A_{k+1..j}$, plus le coût de la multiplication de ces deux matrices. Si l'on se rappelle que chaque matrice A_i est de dimension $p_{i-1} \times p_i$, on voit que le calcul de la matrice produit $A_{i..k} A_{k+1..j}$ demande $p_{i-1} p_k p_j$ multiplications scalaires. On obtient donc

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

Cette équation récursive suppose que l'on connaisse la valeur de k , ce qui n'est pas le cas. Cela dit, il n'existe que $j - i$ valeurs possibles pour k , à savoir $k = i, i + 1, \dots, j - 1$. Comme le parenthésage optimal doit utiliser l'une de ces valeurs pour k , il suffit de toutes les vérifier pour trouver la meilleure. Notre définition récursive du coût minimal de parenthésage du produit $A_i A_{i+1} \cdots A_j$ devient

$$m[i, j] = \begin{cases} 0 & \text{si } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{si } i < j. \end{cases} \quad (15.12)$$

Les valeurs $m[i, j]$ donnent les coûts des solutions optimales des sous-problèmes. Pour mieux comprendre la construction d'une solution optimale, appelons $s[i, j]$ une valeur de k à laquelle on peut fractionner le produit $A_i A_{i+1} \cdots A_j$ pour obtenir un parenthésage optimal. Autrement dit, $s[i, j]$ est égal à une valeur de k telle que $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

d) Calcul des coûts optimaux

À ce stade, c'est chose simple que d'écrire un algorithme récursif basé sur la récurrence (15.12) pour calculer le coût minimal $m[1, n]$ de la multiplication $A_1 A_2 \cdots A_n$. Toutefois, comme nous le verrons à la section 15.3, cet algorithme prend un temps exponentiel, ce qui n'est pas mieux que la méthode primitive consistant à tester chaque manière de parenthéser le produit.

L'observation importante que nous pouvons faire à ce stade est que le nombre de sous-problèmes est assez réduit : un problème pour chaque choix de i et j satisfaisant à $1 \leq i \leq j \leq n$, soit au total $\binom{n}{2} + n = \Theta(n^2)$. Un algorithme récursif peut rencontrer chaque sous-problème plusieurs fois dans différentes branches de son arbre de récursivité. Cette propriété de chevauchement des sous-problèmes est le deuxième grand critère de l'applicabilité de la programmation dynamique (le premier étant la sous-structure optimale).

Au lieu de calculer la solution de la récurrence (15.12) récursivement, on passe donc à la troisième étape de la programmation dynamique en calculant le coût optimal via une approche tabulaire ascendante. Le pseudo code suivant suppose que la matrice A_i est de dimensions $p_{i-1} \times p_i$, pour $i = 1, 2, \dots, n$. L'entrée est une séquence $\langle p_0, p_1, \dots, p_n \rangle$, où $\text{longueur}[p] = n + 1$. La procédure utilise un tableau auxiliaire $m[1 \dots n, 1 \dots n]$ pour mémoriser les coûts $m[i, j]$ et un tableau auxiliaire $s[1 \dots n, 1 \dots n]$ qui mémorise quel est l'indice de k qui avait donné le coût optimal lors du calcul de $m[i, j]$.

Pour implémenter correctement l'approche ascendante (bottom-up), nous devons déterminer quels sont les éléments du tableau qui servent à calculer $m[i, j]$. L'équation (15.12) montre que le coût $m[i, j]$ du calcul d'un produit enchaîné de $j - i + 1$ matrices ne dépend que des coûts de calcul de produits enchaînés impliquant moins de $j - i + 1$ matrices. Autrement dit, pour $k = i, i + 1, \dots, j - 1$, la matrice $A_{i..k}$ est un produit de $k - i + 1 < j - i + 1$ matrices et la matrice $A_{k+1..j}$ est un produit de $j - k < j - i + 1$ matrices. L'algorithme doit donc remplir le tableau m d'une façon qui corresponde à la résolution du problème du parenthésage pour des chaînes de matrices de longueur croissante.

ORDRE-CHAÎNE-MATRICES(p)

```

1   $n \leftarrow \text{longueur}[p] - 1$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire  $m[i, i] \leftarrow 0$ 
```

```

4  pour  $l \leftarrow 2$  à  $n$            ▷  $l$  est la longueur de la chaîne.
5      faire pour  $i \leftarrow 1$  à  $n - l + 1$ 
6          faire  $j \leftarrow i + l - 1$ 
7               $m[i, j] \leftarrow \infty$ 
8              pour  $k \leftarrow i$  à  $j - 1$ 
9                  faire  $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
10                     si  $q < m[i, j]$ 
11                         alors  $m[i, j] \leftarrow q$ 
12                              $s[i, j] \leftarrow k$ 
13  retourner  $m$  et  $s$ 

```

L'algorithme commence par l'affectation $m[i, i] \leftarrow 0$, pour $i = 1, 2, \dots, n$ (coûts minimaux pour les chaînes de longueur 1) aux lignes 2–3. Il utilise ensuite la récurrence (15.12) pour calculer $m[i, i+1]$ pour $i = 1, 2, \dots, n-1$ (coûts minimaux pour les chaînes de longueur $l = 2$) pendant la première exécution de la boucle des lignes 4–12. Au deuxième passage dans la boucle, il calcule $m[i, i+2]$ pour $i = 1, 2, \dots, n-2$ (coûts minimaux pour les chaînes de longueur $l = 3$), et ainsi de suite. A chaque étape, le coût $m[i, j]$ calculé aux lignes 9–12 ne dépend que des éléments de tableau $m[i, k]$ et $m[k + 1, j]$ déjà calculés.

La figure 15.3 illustre cette procédure pour une suite de $n = 6$ matrices. Comme nous n'avons défini $m[i, j]$ que pour $i \leq j$, seule la partie du tableau m strictement supérieure à la diagonale principale est utilisée. La figure présente le tableau de façon à faire apparaître la diagonale principale horizontalement. La chaîne de matrices est donnée en bas. D'après ce modèle, le coût minimal $m[i, j]$ pour la multiplication d'une sous-chaîne $A_i A_{i+1} \cdots A_j$ de matrices peut être trouvé à l'intersection des lignes partant de A_i vers le Nord-Est, et de A_j vers le Nord-Ouest. Chaque ligne horizontale du tableau contient les éléments pour les chaînes de matrices de même longueur. ORDRE-CHAÎNE-MATRICES calcule les lignes du bas vers le haut, et de gauche à droite à l'intérieur de chaque ligne. Un élément $m[i, j]$ est calculé à l'aide des produits $p_{i-1}p_kp_j$ pour $k = i, i+1, \dots, j-1$ et tous les éléments situés au Sud-Ouest et au Sud-Est de $m[i, j]$.

Un examen simple de la structure de boucles imbriquées de ORDRE-CHAÎNE-MATRICES donne un temps d'exécution $O(n^3)$ pour l'algorithme. Les boucles sont imbriquées sur trois niveaux, et chaque indice de boucle (l, i et k) prend au plus $n - 1$ valeurs. L'exercice 15.2.4 vous demandera de montrer que le temps d'exécution de cet algorithme est, en fait, aussi $\Omega(n^3)$. L'algorithme nécessite un espace de stockage $\Theta(n^2)$ pour les tableaux m et s . ORDRE-CHAÎNE-MATRICES est donc beaucoup plus efficace que la méthode en temps exponentiel consistant à énumérer tous les parenthésages possibles et à tester chacun d'eux.

e) Construction d'une solution optimale

Bien que ORDRE-CHAÎNE-MATRICES détermine le nombre optimal de multiplications scalaires nécessaires pour calculer le produit d'une suite de matrices, elle ne

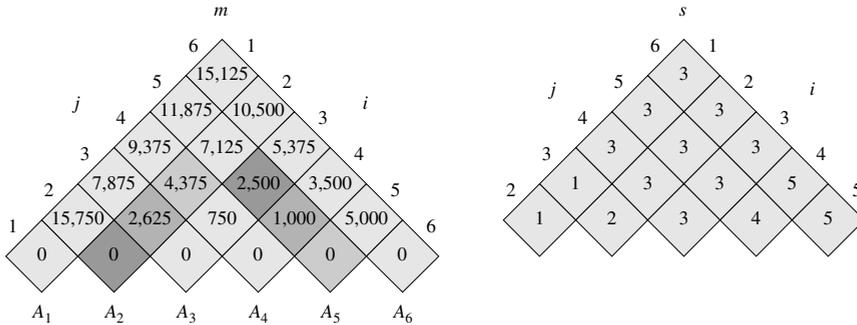


Figure 15.3 Les tableaux m et s calculés par ORDRE-CHAÎNE-MATRICES pour $n = 6$ et les dimensions de matrices suivantes :

matrice	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Les tableaux sont tournés pour présenter horizontalement leur diagonale principale. Seule la diagonale principale et le triangle supérieur sont utilisés dans le tableau m ; pour le tableau s , seul le triangle supérieur est utilisé. Le nombre minimal de multiplications scalaires nécessaires pour multiplier les 6 matrices est $m[1, 6] = 15\,125$. Parmi les éléments foncés, les paires ayant le même ombrage sont regroupées en ligne 9 lors du calcul de

$$\begin{aligned}
 m[2, 5] = \min & \begin{cases} m[2, 2] + m[3, 5] + \rho_1 \rho_2 \rho_5 = 0 + 2\,500 + 35 \cdot 15 \cdot 20 = 13\,000, \\ m[2, 3] + m[4, 5] + \rho_1 \rho_3 \rho_5 = 2\,625 + 1\,000 + 35 \cdot 5 \cdot 20 = 7\,125, \\ m[2, 4] + m[5, 5] + \rho_1 \rho_4 \rho_5 = 4\,375 + 0 + 35 \cdot 10 \cdot 20 = 11\,375 \end{cases} \\
 & = 7\,125.
 \end{aligned}$$

montre pas directement comment multiplier les matrices. Il n'est pas difficile de construire une solution optimale à partir des données calculées et mémorisées dans le tableau $s[1..n, 1..n]$. Chaque élément $s[i, j]$ contient la valeur de k telle que le parenthésage optimal de $A_i A_{i+1} \cdots A_j$ implique un fractionnement du produit entre A_k et A_{k+1} . On sait donc que, pour le calcul optimal de $A_{1..n}$, la dernière multiplication matricielle sera le produit de $A_{1..s[1,n]}$ et $A_{s[1,n]+1..n}$. Les multiplications antérieures peuvent être calculées récursivement; en effet, $s[1, s[1, n]]$ détermine la dernière multiplication effectuée lors du calcul de $A_{1..s[1,n]}$ et $s[s[1, n] + 1, n]$ détermine la dernière multiplication effectuée lors du calcul de $A_{s[1,n]+1..n}$. La procédure récursive ci-après affiche un parenthésage optimal de $\langle A_i, A_{i+1}, \dots, A_j \rangle$, à partir du tableau s calculé par ORDRE-CHAÎNE-MATRICES et des indices i et j . L'appel initial AFFICHE-PARENTHÉSAGE-OPTIMAL($s, 1, n$) affiche un parenthésage optimal de $\langle A_1, A_2, \dots, A_n \rangle$.

```

AFFICHAGE-PARENTHÉSAGE-OPTIMAL( $s, i, j$ )
1  si  $i = j$ 
2    alors afficher«  $A$  »;
3    sinon afficher« ( «
4      AFFICHAGE-PARENTHÉSAGE-OPTIMAL( $s, i, s[i, j]$ )
5      AFFICHAGE-PARENTHÉSAGE-OPTIMAL( $s, s[i, j] + 1, j$ )
6      print » ) »

```

Dans l'exemple de la figure 15.3, l'appel AFFICHAGE-PARENTHÉSAGE-OPTIMAL($s, 1, 6$) affiche le parenthésage $((A_1(A_2A_3))(A_4A_5)A_6)$.

Exercices

15.2.1 Trouver un parenthésage optimal pour le produit d'une suite de matrices dont les dimensions sont données par la séquence $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2.2 Donner un algorithme récursif MULTIPLICATION-CHAÎNE-MATRICES(A, s, i, j) qui effectue les multiplications matricielles enchaînées proprement dites, et ce à partir de la chaîne de matrices $\langle A_1, A_2, \dots, A_n \rangle$, du tableau s calculé par ORDRE-CHAÎNE-MATRICES, et des indices i et j . (L'appel initial sera MULTIPLICATION-CHAÎNE-MATRICES($A, s, 1, n$).)

15.2.3 Employer la méthode de substitution pour montrer que la solution de la récurrence (15.11) est $\Omega(2^n)$.

15.2.4 Soit $R(i, j)$ le nombre de fois que l'élément de tableau $m[i, j]$ est référencé pendant le calcul d'autres éléments de tableau dans un appel à ORDRE-CHAÎNE-MATRICES. Montrer que le nombre total de références pour le tableau tout entier est

$$\sum_{i=1}^n \sum_{j=i}^n R(i, j) = \frac{n^3 - n}{3}.$$

(Conseil : L'équation (A.3) pourra vous être utile.)

15.2.5 Montrer qu'un parenthésage entier d'une expression à n éléments contient exactement $n - 1$ paires de parenthèses.

15.3 ÉLÉMENTS DE LA PROGRAMMATION DYNAMIQUE

Bien que vous ayez déjà travaillé sur deux exemples, vous vous demandez peut-être dans quelles situations appliquer la programmation dynamique. Dans la perspective de l'ingénierie, quand faut-il envisager une solution à base de programmation dynamique ? Dans cette section, nous allons examiner les deux grandes caractéristiques que doit posséder un problème d'optimisation pour que la programmation dynamique

soit applicable : sous-structure optimale et chevauchement des sous-problèmes. Nous étudierons également une variante, baptisée recensement (mémoïsation), qui permet d'exploiter la propriété de chevauchement des sous-problèmes.

a) *Sous-structure optimale*

La première étape de la résolution d'un problème d'optimisation via programmation dynamique est de caractériser la structure d'une solution optimale. Retenons qu'un problème exhibe une **sous-structure optimale** si une solution optimale au problème contient en elle des solutions optimales de sous-problèmes. Chaque fois qu'un problème exhibe une sous-structure optimale, c'est un bon indice de l'utilisabilité de la programmation dynamique. (Cela peut aussi signifier qu'une stratégie gloutonne est applicable. Voir chapitre 16.) Avec la programmation dynamique, on construit une solution optimale du problème à partir de solutions optimales de sous-problèmes. Par conséquent, on doit penser à vérifier que la gamme des sous-problèmes que l'on considère inclut les sous-problèmes utilisés dans une solution optimale.

Nous avons découvert la sous-structure optimale dans les deux problèmes étudiés jusqu'ici dans ce chapitre. À la section 15.1, nous avons observé que le chemin optimal passant par le poste j de l'une ou l'autre des chaînes de montage contenait le chemin optimal passant par le poste $j - 1$ d'une des deux chaînes. À la section 15.2, nous avons observé qu'un parenthésage optimal de $A_i A_{i+1} \cdots A_j$ qui fractionne le produit de A_k et A_{k+1} contient des solutions optimales pour les problèmes consistant à parenthésier $A_i A_{i+1} \cdots A_k$ et $A_{k+1} A_{k+2} \cdots A_j$.

La découverte de la sous-structure optimale obéit au schéma général suivant :

- 1) Vous montrez qu'une solution du problème consiste à faire un choix, par exemple à choisir un poste précédant dans une chaîne de montage ou un emplacement de fractionnement dans une chaîne de produits de matrices. Ayant fait ce choix, vous êtes ramené à résoudre un ou plusieurs sous-problèmes.
- 2) Vous supposez que, pour un problème donné, on vous donne le choix qui conduit à une solution optimale. Pour l'instant, vous ne vous souciez pas de la façon dont on détermine ce choix. Vous faites comme si on vous le donnait tout cuit.
- 3) À partir de ce choix, vous déterminez quels sont les sous-problèmes qui en découlent et comment caractériser au mieux l'espace des sous-problèmes résultant.
- 4) Vous montrez que les solutions des sous-problèmes employées par la solution optimale du problème doivent elles-mêmes être optimales, et ce en utilisant la technique du « couper-coller » : vous supposez que chacune des solutions de sous-problème n'est pas optimale et vous en déduisez une contradiction. En particulier, en « coupant » une solution de sous-problème non optimale et en la « collant » dans la solution optimale, vous montrez que vous obtenez une meilleure solution pour le problème initial, ce qui contredit l'hypothèse que vous avez déjà une solution optimale. S'il y a plusieurs sous-problèmes, ils sont généralement

similaires, de sorte que l'argument couper-coller utilisé pour l'un peut resservir pour les autres, moyennant une petite adaptation.

Pour caractériser l'espace des sous-problèmes, une règle empirique consiste à essayer de garder l'espace aussi simple que possible puis à l'étendre en fonction des besoins. Ainsi, l'espace des sous-problèmes que nous avons considéré pour l'ordonnement des chaînes de montage était le chemin optimal entre l'entrée de l'atelier et les postes $S_{1,j}$ et $S_{2,j}$. Cet espace de sous-problèmes fonctionnait bien, et il n'était point besoin d'essayer un espace plus général de sous-problèmes.

Inversement, supposons que nous ayons essayé de limiter notre espace de sous-problèmes pour la multiplication de matrices en chaîne à des produits matriciels de la forme $A_1A_2 \cdots A_j$. Comme précédemment, un parenthésage optimal doit fractionner ce produit entre A_k et A_{k+1} pour un certain $1 \leq k \leq j$. À moins que nous puissions garantir que k est toujours égal à $j - 1$, nous trouverons que nous avons des sous-problèmes de la forme $A_1A_2 \cdots A_k$ et $A_{k+1}A_{k+2} \cdots A_j$, et que ce dernier sous-problème n'est pas de la forme $A_1A_2 \cdots A_j$. Pour ce problème, il fallait permettre aux sous-problèmes de varier « aux deux bouts », c'est-à-dire permettre à i et à j de varier tous les deux dans le sous-problème $A_iA_{i+1} \cdots A_j$.

La sous-structure optimale varie d'un domaine de problème à l'autre de deux façons :

- 1) le nombre de sous-problèmes qui sont utilisés dans une solution optimale du problème originel, et
- 2) le nombre de choix que l'on a pour déterminer le(s) sous-problème(s) à utiliser dans une solution optimale.

Dans l'ordonnement de chaîne de montage, une solution optimale n'utilise qu'un seul sous-problème, mais il faut considérer deux choix pour déterminer une solution optimale. Pour trouver le chemin optimal passant par le poste $S_{i,j}$, on utilise *soit* le chemin optimal passant par $S_{1,j-1}$ *soit* le chemin optimal passant par $S_{2,j-1}$; quel que soit le choix que nous ferons, il représentera le sous-problème à résoudre de manière optimale. La multiplication enchaînée de matrices pour la sous-chaîne $A_iA_{i+1} \cdots A_j$ est un exemple à deux sous-problèmes et $j - i$ choix. Pour une matrice donnée A_k au niveau de laquelle on fractionne le produit, on a deux sous-problèmes (parenthéser $A_iA_{i+1} \cdots A_k$ et parenthéser $A_{k+1}A_{k+2} \cdots A_j$) et l'on doit résoudre *les deux* de manière optimale. Après avoir déterminé les solutions optimales des sous-problèmes, on choisit parmi $j - i$ candidats pour l'indice k .

De manière informelle, le temps d'exécution d'un algorithme de programmation dynamique dépend du produit de deux facteurs : le nombre de sous-problèmes et le nombre de choix envisagés pour chaque sous-problème. Dans l'ordonnement des chaînes de montage, on avait $\Theta(n)$ sous-problèmes et seulement deux choix possibles pour chaque sous-problème, ce qui donnait un temps d'exécution $\Theta(n)$. Pour la multiplication de matrices en chaîne, il y avait $\Theta(n^2)$ sous-problèmes et chaque sous-problème proposait au plus $n - 1$ choix, ce qui donnait un temps d'exécution $O(n^3)$.

La programmation dynamique utilise la sous-structure optimale de manière ascendante (bottom-up) : on commence par trouver des solutions optimales pour les sous-problèmes, après quoi l'on trouve une solution optimale pour le problème. Trouver une solution optimale pour le problème implique de faire un choix entre des sous-problèmes : lesquels prendre pour résoudre le problème ? Le coût de la solution du problème est généralement le cumul des coûts des sous-problèmes, plus un coût lié directement aux choix lui-même. Dans l'ordonnement des chaînes de montage, par exemple, on commençait par résoudre les sous-problèmes consistant à trouver le chemin optimal passant par les postes $S_{1,j-1}$ et $S_{2,j-1}$; ensuite, on choisissait l'un de ces postes comme poste précédant le poste $S_{i,j}$. Le coût attribuable au choix lui-même dépendait de ce que l'on changeait ou non de chaîne de montage entre les postes $j-1$ et j ; ce coût était $a_{i,j}$ si l'on restait sur la même chaîne et $t_{i',j-1} + a_{i,j}$, avec $i' \neq i$, si l'on changeait de chaîne. Dans la multiplication des matrices, on déterminait des parenthésages optimaux pour les sous-chaînes de $A_i A_{i+1} \cdots A_j$, puis l'on choisissait la matrice au niveau de laquelle il fallait fractionner le produit. Le coût attribuable au choix lui-même était donné par le terme $p_{i-1} p_k p_j$.

Au chapitre 16, nous étudierons les « algorithmes gloutons », qui présentent moult similitudes avec la programmation dynamique. En particulier, les problèmes auxquels s'appliquent des algorithmes gloutons ont une sous-structure optimale. Une grande différence entre algorithmes gloutons et programmation dynamique est que, avec les algorithmes gloutons, on utilise la sous-structure optimale d'une manière descendante (top-down). Au lieu de commencer par trouver des solutions optimales pour les sous-problèmes puis de choisir un sous-problème, un algorithme glouton commence par faire un choix (celui qui semble être le meilleur sur le moment) puis résout le sous-problème résultant.

► Subtilités

Il ne faut pas supposer qu'une sous-structure optimale est applicable quand ce n'est pas le cas. Considérons les deux problèmes suivants, dans lesquels on a un graphe orienté $G = (V, E)$ et des sommets $u, v \in V$.

Plus court chemin non pondéré ⁽¹⁾ : Trouver un chemin entre u et v qui soit composé d'un minimum d'arcs. Un tel chemin doit être élémentaire, car le fait de supprimer un circuit d'un chemin produit un circuit ayant moins d'arcs.

Plus long chemin élémentaire non pondéré : Trouver un chemin élémentaire de u à v qui soit composé du plus grand nombre d'arcs possible. Il faut ajouter la contrainte de chemin élémentaire, car autrement on peut traverser un circuit autant de fois que l'on veut pour créer un chemin ayant un nombre d'arcs arbitraire.

(1) Nous employons le terme « non pondéré » pour distinguer ce problème de celui consistant à trouver des plus courts chemins composés d'arcs pondérés, que nous verrons aux chapitres 24 et 25. La technique de recherche en largeur, traitée au chapitre 22, permet de résoudre le problème non pondéré.

Le problème du plus court chemin non pondéré exhibe une sous-structure optimale, comme nous allons le voir. Supposons que $u \neq v$, afin que le problème soit non trivial. Alors, tout chemin p de u à v contient un sommet intermédiaire, par exemple w . (Notez que w peut être u ou v .) Donc, on peut décomposer le chemin $u \xrightarrow{p} v$ en sous-chemins $u \xrightarrow{p_1} w \xrightarrow{p_2} v$. Visiblement, le nombre d'arcs de p est la somme du nombre d'arcs de p_1 et du nombre d'arcs de p_2 . Nous affirmons que, si p est un chemin optimal (c'est-à-dire, un plus court chemin) de u à v , alors p_1 est un plus court chemin de u à w . Pourquoi ? Nous utilisons un raisonnement basé sur la technique du « couper-coller » : s'il y avait un autre chemin, disons p'_1 , de u à w ayant moins d'arcs que p_1 , alors on pourrait couper p_1 et coller p'_1 pour produire un chemin $u \xrightarrow{p'_1} w \xrightarrow{p_2} v$ ayant moins d'arcs que p , ce qui contredirait l'optimalité de p . De manière symétrique, p_2 est un plus court chemin de w à v . Donc, on peut trouver un plus court chemin de u à v en considérant tous les sommets intermédiaires w , en trouvant un plus court chemin de u à w et un plus court chemin de w à v , puis en choisissant un sommet intermédiaire w qui donne le plus court chemin globalement. À la section 25.2, nous emploierons une variante de cette observation de la sous-structure optimale pour trouver un plus court chemin entre chaque paire de sommets d'un graphe orienté pondéré.

Il est tentant de penser que le problème consistant à trouver un plus long chemin élémentaire non pondéré exhibe une sous-structure optimale, lui aussi. Après tout, si l'on décompose un plus long chemin élémentaire $u \xrightarrow{p} v$ en sous-chemins $u \xrightarrow{p_1} w \xrightarrow{p_2} v$, alors p_1 n'est-il pas un plus long chemin élémentaire de u à w et p_2 n'est-il pas un plus long chemin élémentaire de w à v ? La réponse est non ! La figure 15.4 donne un exemple. Considérons le chemin $q \rightarrow r \rightarrow t$, qui est un plus long chemin élémentaire de q à t . Est-ce que $q \rightarrow r$ est un plus long chemin élémentaire de q à r ? Non, car le chemin $q \rightarrow s \rightarrow t \rightarrow r$ est un chemin élémentaire qui est plus long. Est-ce que $r \rightarrow t$ est un plus long chemin élémentaire de r à t ? Non, car le chemin $r \rightarrow q \rightarrow s \rightarrow t$ est un chemin élémentaire qui est plus long.

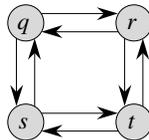


Figure 15.4 Graphe orienté montrant que le problème qui consiste à trouver un plus long chemin élémentaire dans un graphe orienté non pondéré n'a pas de sous-structure optimale. Le chemin $q \rightarrow r \rightarrow t$ est un plus long chemin élémentaire de q à t , mais le sous-chemin $q \rightarrow r$ n'est pas un plus long chemin élémentaire de q à r , pas plus que le sous-chemin $r \rightarrow t$ n'est un plus long chemin élémentaire de r à t .

Cet exemple montre que, pour les plus longs chemins élémentaires, non seulement il manque une sous-structure optimale, mais en plus on ne peut pas toujours construire une solution « licite » à partir de solutions de sous-problèmes. Si l'on combine les plus longs chemins élémentaires $q \rightarrow s \rightarrow t \rightarrow r$ et $r \rightarrow q \rightarrow s \rightarrow t$, on obtient le

chemin $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$ qui n'est pas élémentaire. En fait, le problème consistant à trouver un plus long chemin élémentaire non pondéré ne semble pas avoir de sous-structure optimale de quelque sorte que ce soit. Aucun algorithme efficace de programmation dynamique n'a pu être trouvé pour ce problème. En fait, ce problème est NP-complet (voir chapitre 34), ce qui implique qu'il est peu probable qu'il puisse être résolu en temps polynomial.

Qu'est-ce qui fait que la sous-structure d'un plus long chemin élémentaire est si différente de celle d'un plus court chemin ? Deux sous-problèmes sont utilisés dans une solution du problème des plus longs ou des plus courts chemins, mais les sous-problèmes du problème du plus long chemin élémentaire ne sont pas *indépendants* alors que ceux du problème du plus court chemin le sont. Qu'entend-on par sous-problèmes indépendants ? On entend que la solution d'un sous-problème n'affecte pas la solution d'un autre sous-problème du même problème. Dans l'exemple de la figure 15.4, le problème consiste à trouver un plus long chemin élémentaire de q à t ayant deux sous-problèmes : trouver des plus longs chemins élémentaires de q à r et de r à t . Pour le premier de ces sous-problèmes, on choisit le chemin $q \rightarrow s \rightarrow t \rightarrow r$, et donc on a utilisé les sommets s et t . On ne peut plus utiliser ces sommets dans le second sous-problème, vu que la combinaison des deux solutions de sous-problèmes donnerait un chemin qui n'est pas élémentaire. Si l'on ne peut pas utiliser le sommet t dans le second problème, alors on ne peut pas résoudre du tout, car il faut que t soit sur le chemin que nous trouvons et ce n'est pas le sommet au niveau duquel nous « recollons » les solutions des sous-problème (ce sommet est r). Le fait que nous utilisions les sommets s et t dans une solution de sous-problème nous empêche de les utiliser dans l'autre solution de sous-problème. Or, nous devons utiliser au moins l'un des deux pour résoudre l'autre sous-problème, et nous devons utiliser les deux pour le résoudre de manière optimale. Donc, nous disons que ces sous-problèmes ne sont pas indépendants. Vu d'une autre façon, l'utilisation de ressources pour la résolution d'un sous-problème (ces ressources étant des sommets) les rend indisponibles pour l'autre sous-problème.

Qu'est-ce qui fait, alors, que les sous-problèmes sont indépendants dans le problème du plus court chemin ? La réponse est que, par nature, les sous-problèmes n'ont pas de ressources en commun. Nous affirmons que, si un sommet w est sur un plus court chemin p de u à v , alors on peut coller *n'importe quel* plus court chemin $u \overset{p_1}{\rightsquigarrow} w$ avec *n'importe quel* plus court chemin $w \overset{p_2}{\rightsquigarrow} v$ pour produire un plus court chemin de u à v . Nous sommes assurés que, à part w , aucun sommet ne peut apparaître à la fois sur les deux chemins p_1 et p_2 . Pourquoi ? Supposez qu'un sommet $x \neq w$ appartienne à la fois à p_1 et p_2 ; on peut alors décomposer p_1 en $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ et p_2 en $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. En vertu de la sous-structure optimale de ce problème, le chemin p a autant d'arcs que p_1 et p_2 réunis ; disons que p a e arcs. Maintenant, construisons un chemin $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ de u à v . Ce chemin a au plus $e - 2$ arcs, ce qui contredit l'hypothèse que p est un plus court chemin. Donc, nous avons la certitude que les sous-problèmes du problème du plus court chemin sont indépendants. Les deux problèmes traités aux sections 15.1

et 15.2 ont des sous-problèmes indépendants. Dans la multiplication matricielle en chaîne, les sous-problèmes consistent à multiplier les sous-chaînes $A_i A_{i+1} \cdots A_k$ et $A_{k+1} A_{k+2} \cdots A_j$. Ces sous-chaînes sont disjointes, de sorte qu'aucune matrice ne peut être dans les deux. Dans l'ordonnement de chaîne de montage, pour déterminer le chemin optimal passant par le poste $S_{i,j}$, on regarde les chemins optimaux passant par les postes $S_{1,j-1}$ et $S_{2,j-1}$. Comme notre solution pour le chemin optimal passant par le poste $S_{i,j}$ inclut une seule des solutions de sous-problème, ce sous-problème est automatiquement indépendant de tous les autres sous-problèmes employés dans la solution.

b) Chevauchement des sous-problèmes

La seconde caractéristique que doit avoir un problème d'optimisation pour que la programmation dynamique lui soit applicable est la suivante : l'espace des sous-problèmes doit être « réduit », au sens où un algorithme récursif pour le problème résout constamment les mêmes sous-problèmes au lieu d'en engendrer toujours de nouveaux. En général, le nombre total de sous-problèmes distincts est polynomial par rapport à la taille de l'entrée. Quand un algorithme récursif repasse sur le même problème constamment, on dit que le problème d'optimisation contient des **sous-problèmes qui se chevauchent**. *A contrario*, un problème pour lequel l'approche diviser-pour-régner est plus adaptée génère, le plus souvent, des problèmes nouveaux à chaque étape de la récursivité. Les algorithmes de programmation dynamique tirent parti du chevauchement des sous-problèmes en résolvant chaque sous-problème une fois, puis en stockant la solution dans un tableau, ce qui permettra ultérieurement de retrouver la solution avec un temps de recherche constant.

À la section 15.1, nous avons vu brièvement comment une solution récursive de l'ordonnement de chaîne de montage fait 2^{n-j} références à $f_i[j]$, avec $j = 1, 2, \dots, n$. Notre solution tabulaire permet de réduire un algorithme récursif à temps exponentiel à un temps linéaire.

Pour illustrer de manière plus précise la propriété de chevauchement des sous-problèmes, réexaminons le problème de la multiplication de matrices en chaîne. Sur la figure 15.3, on remarque que ORDRE-CHAÎNE-MATRICES recherche itérativement la solution de sous-problèmes situés sur des lignes plus basses quand elle résout les sous-problèmes situés sur des lignes plus hautes. Par exemple, l'élément $m[3, 4]$ est référencé 4 fois : pendant les calculs de $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ et $m[3, 6]$. Si $m[3, 4]$ était recalculé chaque fois au lieu d'être lu dans une table, le temps d'exécution augmenterait très rapidement. Pour comprendre cela, considérons la procédure récursive (inefficace) ci-après, qui détermine $m[i, j]$, nombre minimal de multiplications scalaires nécessaires pour calculer le produit d'une suite de matrices $A_{i..j} = A_i A_{i+1} \cdots A_j$. La procédure est directement basée sur la récurrence (15.12).

CHAÎNE-MATRICES-RÉCURSIF(p, i, j)

- 1 si $i = j$
- 2 **alors retourner** 0
- 3 $m[i, j] \leftarrow \infty$

```

4  pour  $k \leftarrow i$  à  $j - 1$ 
5    faire  $q \leftarrow$  CHAÎNE-MATRICES-RÉCURSIF( $p, i, k$ )
           + CHAÎNE-MATRICES-RÉCURSIF( $p, k + 1, j$ )
           +  $p_{i-1}p_kp_j$ 
6    si  $q < m[i, j]$ 
7    alors  $m[i, j] \leftarrow q$ 
8  retourner  $m[i, j]$ 

```

La figure 15.5 montre l'arborescence récursive correspondant à CHAÎNE-MATRICES-RÉCURSIF($p, 1, 4$). Chaque nœud est étiqueté par les valeurs des paramètres i et j . Notez que certaines paires de valeurs apparaissent plusieurs fois.

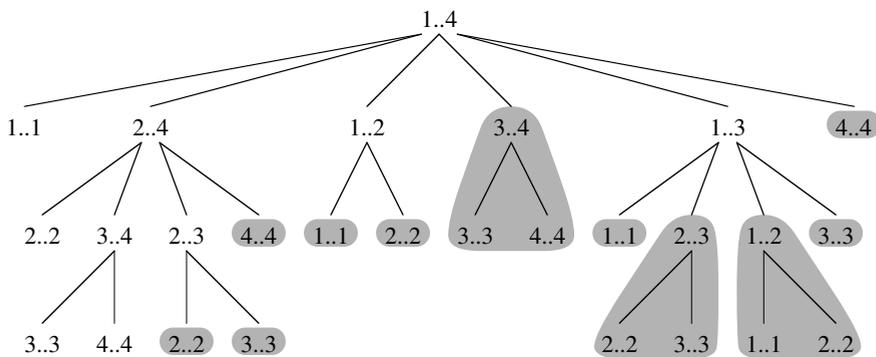


Figure 15.5 Arborescence récursive correspondant au calcul de CHAÎNE-MATRICES-RÉCURSIF($p, 1, 4$). Chaque nœud contient les paramètres i et j . La procédure MÉMORISATION-CHAÎNE-MATRICES($p, 1, 4$) remplace par une lecture dans un tableau les calculs qui sont effectués dans un sous-arbre sur fond gris.

En fait, on peut montrer que le temps d'exécution requis par cette procédure récursive pour calculer $m[1, n]$ est au moins exponentiel en n . Soit $T(n)$ le temps mis par CHAÎNE-MATRICES-RÉCURSIF pour calculer un parenthésage optimal d'une chaîne de n matrices. Si l'on suppose que l'exécution des lignes 1–2 et des lignes 6–7 prend chacune au moins une unité de temps, l'analyse de la procédure conduit à la récurrence

$$T(1) \geq 1,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{pour } n > 1.$$

En remarquant que, pour $i = 1, 2, \dots, n-1$, chaque terme $T(i)$ apparaît une fois sous la forme $T(k)$ et une fois sous la forme $T(n-k)$, puis en regroupant les $n-1$ termes 1 de la sommation avec le 1 extérieur, on peut réécrire la récurrence sous la forme :

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n. \quad (15.13)$$

Nous allons prouver que $T(n) = \Omega(2^n)$ à l'aide de la méthode de substitution. Plus précisément, nous allons montrer que $T(n) \geq 2^{n-1}$ pour tout $n \geq 1$. La base est simple à établir, puisque $T(1) \geq 1 = 2^0$. Par récurrence, pour $n \geq 2$, on a

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\geq 2^{n-1}, \end{aligned}$$

ce qui achève la démonstration. Donc, la quantité totale de travail effectué par l'appel CHAÎNE-MATRICES-RÉCURSIF($p, 1, n$) est au moins exponentielle en n .

Comparons cet algorithme récursif descendant (top-down) à l'algorithme de programmation dynamique ascendant. Ce dernier est plus efficace parce qu'il tire parti de la propriété de superposition des sous-problèmes. Il n'existe que $\Theta(n^2)$ sous-problèmes différents, et l'algorithme de programmation dynamique résout chaque sous-problème une seule fois. En revanche, l'algorithme récursif recommence la résolution d'un sous-problème chaque fois que celui-ci réapparaît dans l'arborescence récursive. Chaque fois qu'une arborescence récursive traduisant la solution récursive naïve d'un problème contient régulièrement le même sous-problème et que le nombre total des sous-problèmes est petit, c'est une bonne idée que de penser à une résolution *via* programmation dynamique.

c) Reconstruction d'une solution optimale

Concrètement, on stocke souvent dans un tableau le choix que l'on a fait pour chaque sous-problème ; cela nous dispense de reconstruire cette donnée à partir des coûts que l'on a stockés. Dans l'ordonnancement de chaîne de montage, on stockait dans $l_i[j]$ le poste précédant $S_{i,j}$ dans un chemin optimal passant par $S_{i,j}$. Une autre technique serait la suivante : après remplissage du tableau $f_i[j]$ tout entier, on détermine quel est le poste qui précède $S_{1,j}$ dans un chemin optimal passant par $S_{i,j}$, et ce avec un peu de travail supplémentaire. Si $f_1[j] = f_1[j-1] + a_{1,j}$, alors le poste $S_{1,j-1}$ précède $S_{1,j}$ dans un chemin optimal passant par $S_{1,j}$. Autrement, c'est que $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ et donc que $S_{2,j-1}$ précède $S_{1,j}$. Pour l'ordonnancement de chaîne de montage, reconstruire les postes prédécesseur prend seulement un temps $O(1)$ par poste, même sans le tableau $l_i[j]$.

Pour le produit de matrices en chaîne, en revanche, le tableau $s[i, j]$ nous épargne un travail significatif quand on reconstruit une solution optimale. Supposons que l'on ne gère pas de tableau $s[i, j]$ et que l'on se contente de gérer le tableau $m[i, j]$ contenant les coûts des sous-problèmes optimaux. Il y a $j - i$ choix pour déterminer quels sont les sous-problèmes à utiliser dans une solution optimale de parenthésage de

$A_i A_{i+1} \cdots A_j$, et $j-i$ n'est pas une constante. Donc, il faudrait un temps $\Theta(j-i) = \omega(1)$ pour reconstruire les sous-problèmes que nous avons choisis pour une solution à un problème donné. En stockant dans $s[i, j]$ l'indice de la matrice au niveau de laquelle on fractionne le produit $A_i A_{i+1} \cdots A_j$, on peut reconstruire chaque choix en temps $O(1)$.

d) Recensement

Il existe une variante de la programmation dynamique qui offre souvent la même efficacité que l'approche usuelle, tout en maintenant une stratégie descendante. Le principe est de **recenser** les actions naturelles, mais inefficaces, de l'algorithme récursif. Comme avec la programmation dynamique ordinaire, on conserve dans un tableau les solutions des sous-problèmes, mais la structure de contrôle pour le remplissage du tableau est plus proche de l'algorithme récursif.

Un algorithme récursif de recensement gère un élément du tableau pour la solution de chaque sous-problème. Chaque élément contient au départ une valeur spéciale, pour indiquer qu'il n'a pas encore été rempli. Lorsque le sous-problème est rencontré pour la première fois durant l'exécution de l'algorithme récursif, sa solution est calculée puis stockée dans le tableau. A chaque réapparition du sous-problème, la valeur stockée dans le tableau est tout simplement lue et retournée au programme principal⁽²⁾.

La procédure suivante est une variante de CHAÎNE-MATRICES-RÉCURSIF avec recensement.

MÉ MORISATION-CHAÎNE-MATRICES(p)

```

1   $n \leftarrow \text{longueur}[p] - 1$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire pour  $j \leftarrow i$  à  $n$ 
4          faire  $m[i, j] \leftarrow \infty$ 
5  retourner RÉCUPÉRER-CHAÎNE( $p, 1, n$ )

```

RÉCUPÉRER-CHAÎNE(p, i, j)

```

1  si  $m[i, j] < \infty$ 
2      alors retourner  $m[i, j]$ 
3  si  $i = j$ 
4      alors  $m[i, j] \leftarrow 0$ 
5      sinon pour  $k \leftarrow i$  à  $j - 1$ 
6          faire  $q \leftarrow$  RÉCUPÉRER-CHAÎNE( $p, i, k$ )
               $+$  RÉCUPÉRER-CHAÎNE( $p, k + 1, j$ )  $+ p_{i-1} p_k p_j$ 
7          si  $q < m[i, j]$ 
8              alors  $m[i, j] \leftarrow q$ 
9  retourner  $m[i, j]$ 

```

(2) Cette approche présuppose que l'ensemble de tous les paramètres possibles de sous-problème est connu, et que la relation entre positions dans le tableau et sous-problèmes est établie. Une autre approche consiste à recenser via hachage, en se servant des paramètres de sous-problème comme clés.

MÉMORISATION-CHAÎNE-MATRICES, à l'instar de ORDRE-CHAÎNE-MATRICES, gère un tableau $m[1..n, 1..n]$ de valeurs calculées de $m[i, j]$, nombre minimal de multiplications scalaires nécessaires pour calculer la matrice $A_{i..j}$. Chaque élément du tableau contient initialement la valeur ∞ pour indiquer que l'élément n'a pas encore été rempli. Lorsque l'appel RÉCUPÉRER-CHAÎNE(p, i, j) est exécuté, si $m[i, j] < \infty$ à la ligne 1, la procédure se contente de retourner le coût $m[i, j]$ calculé précédemment (ligne 2). Sinon, le coût est calculé comme dans CHAÎNE-MATRICES-RÉCURSIF, stocké dans $m[i, j]$ puis retourné au programme principal. (La valeur ∞ est pratique pour désigner un élément non rempli, puisque c'est la valeur utilisée pour initialiser $m[i, j]$ en ligne 3 de CHAÎNE-MATRICES-RÉCURSIF.) Donc, RÉCUPÉRER-CHAÎNE(p, i, j) retourne toujours la valeur de $m[i, j]$ mais ne la calcule que si c'est la première fois que RÉCUPÉRER-CHAÎNE est appelée avec les paramètres i et j .

Sur la figure 15.5, on peut voir l'économie de temps réalisée quand on utilise MÉMORISATION-CHAÎNE-MATRICES au lieu de CHAÎNE-MATRICES-RÉCURSIF. Les arbres sur fond gris représentent les valeurs qui sont relues au lieu d'être calculées.

À l'instar de l'algorithme de programmation dynamique ORDRE-CHAÎNE-MATRICES, la procédure MÉMORISATION-CHAÎNE-MATRICES s'exécute en temps $O(n^3)$. Chacun des $\Theta(n^2)$ éléments du tableau est initialisé une fois à la ligne 4 de MÉMORISATION-CHAÎNE-MATRICES. On peut classer les appels à RÉCUPÉRER-CHAÎNE en deux types :

- 1) appels où $m[i, j] = \infty$, de sorte que les lignes 3–9 sont exécutées, et
- 2) appels où $m[i, j] < \infty$, de sorte que RÉCUPÉRER-CHAÎNE se contente de rendre la main en ligne 2.

Il y a $\Theta(n^2)$ appels du premier type, un par élément de tableau. Tous les appels du second type sont effectués, en tant qu'appels récursifs, par des appels du premier type. Chaque fois qu'un appel de RÉCUPÉRER-CHAÎNE fait des appels récursifs, il en fait $O(n)$. Donc, il y a en tout $O(n^3)$ appels du second type. Chaque appel du second type prend un temps $O(1)$, et chaque appel du premier type prend un temps $O(n)$, plus le temps passé dans ses appels récursifs. Le temps total est donc $O(n^3)$. Le recensement ramène donc un algorithme à temps $\Omega(2^n)$ à un algorithme à temps $O(n^3)$.

En résumé : le problème de la multiplication de matrices en chaîne peut être résolu en temps $O(n^3)$ soit par un algorithme descendant à recensement, soit par un algorithme ascendant de programmation dynamique. Les deux méthodes tirent parti de la propriété de chevauchement des sous-problèmes. Il n'existe au total que $\Theta(n^2)$ sous-problèmes différents, et chacune de ces deux méthodes ne calcule la solution pour chaque sous-problème qu'une seule fois. Sans recensement, l'algorithme récursif basique s'exécute en temps exponentiel, car les sous-problèmes sont résolus de manière répétée.

En général, si tous les sous-problèmes doivent être résolus au moins une fois, normalement un algorithme ascendant de programmation dynamique surpasse d'un facteur constant un algorithme descendant à recensement, car il élimine la surcharge in-

duite par les appels récursifs et diminue la charge de gestion du tableau. Par ailleurs, il existe des problèmes pour lesquels le modèle régulier des accès au tableau, tel que mis en œuvre dans l'algorithme de programmation dynamique, permet de réduire encore plus les besoins en temps et en espace. En revanche, si certains sous-problèmes de l'espace des sous-problèmes n'ont pas besoin d'être résolus du tout, la solution du recensement présente l'avantage de ne résoudre que les sous-problèmes qui sont vraiment nécessaires.

Exercices

15.3.1 Quelle est la manière la plus efficace de déterminer le nombre optimal de multiplications dans un problème de multiplications matricielles en chaîne : énumérer tous les parenthésages possibles dans le produit puis calculer le nombre de multiplications pour chacun, ou bien exécuter CHAÎNE-MATRICES-RÉCURSIF ? Justifier la réponse

15.3.2 Dessiner l'arborescence récursive de la procédure TRI-FUSION, vue à la section 2.3.1, sur un tableau de 16 éléments. Expliquer pourquoi le recensement ne permet pas d'améliorer la vitesse d'un bon algorithme diviser-pour-régner comme TRI-FUSION

15.3.3 Soit une variante du problème des multiplications matricielles en chaîne, dans laquelle on cherche à parenthéser la chaîne de matrices de façon à maximiser, et non à minimiser, le nombre de multiplications scalaires. Ce problème exhibe-t-il une sous-structure optimale ?

15.3.4 Décrire le chevauchement des sous-problèmes dans l'ordonnancement de chaîne de montage.

15.3.5 Nous avons dit qu'avec la programmation dynamique on commence par résoudre les sous-problèmes, après quoi on choisit ceux que l'on va utiliser dans une solution optimale du problème global. Le professeur Folamour affirme qu'il n'est pas toujours indispensable de résoudre tous les sous-problèmes pour trouver une solution optimale. Elle dit que l'on peut trouver une solution optimale pour le problème des produits matriciels en chaîne en choisissant toujours la matrice A_k au niveau de laquelle on fractionne le sous-produit $A_i A_{i+1} \cdots A_j$ (en sélectionnant k de façon qu'il minimise la quantité $p_{i-1} p_k p_j$) avant de résoudre les sous-problèmes. Trouver une instance du problème des multiplications matricielles en chaîne pour laquelle cette approche gloutonne donne une solution sous-optimale.

15.4 PLUS LONGUE SOUS-SÉQUENCE COMMUNE

En biologie, on doit souvent comparer l'ADN de deux (ou plusieurs) organismes. Un échantillon d'ADN est une suite de molécules appelées *bases*, les bases possibles étant l'adénine, la guanine, la cytosine et la thymine. Si l'on représente chacune des ces bases par leurs initiales, un échantillon d'ADN s'exprime sous la forme d'une

chaîne prise sur l'ensemble fini $\{A, C, G, T\}$. (Voir annexe C pour la définition d'une chaîne.) Ainsi, l'ADN d'un organisme sera $S_1 = \text{ACCGGTCGAGTGC GCGGAAGCCGGCCGAA}$ alors que l'ADN d'un autre sera $S_2 = \text{GTCGTTTCGGAATGCCGTTGCTCTGTAAA}$. Comparer deux échantillons d'ADN permet, entre autres, de déterminer leur degré de « similitude », qui mesure en quelque sorte la façon dont les deux organismes sont apparentés. La similitude peut se définir de bien des façons différentes. Par exemple, on peut dire que deux échantillons d'ADN sont semblables si l'un est une sous-chaîne de l'autre. (Le chapitre 32 présente des algorithmes relatifs à cette problématique.) Dans notre exemple, ni S_1 ni S_2 n'est une sous-chaîne de l'autre. Autre possibilité : on pourrait dire que deux échantillons sont semblables si le nombre de modifications nécessaires pour transformer l'un en l'autre est faible. (Le problème 15.3 se penchera sur cette notion.) Un autre moyen de mesurer la similitude des échantillons S_1 et S_2 est de trouver un troisième échantillon S_3 tel que les bases de S_3 apparaissent dans S_1 et dans S_2 ; ces bases doivent apparaître dans le même ordre, mais pas forcément de manière consécutive. Plus l'échantillon S_3 trouvé est long, plus S_1 et S_2 sont semblables. Dans notre exemple, le plus long S_3 est $\text{GTCGTCGGAAGCCGGCCGAA}$.

Nous formaliserons cette dernière notion de similitude sous le nom de problème de la plus longue sous-séquence commune. Une sous-séquence d'une séquence donnée est tout simplement constituée de la séquence de départ, à laquelle on a retiré certains éléments (éventuellement zéro). Plus formellement, étant donnée une séquence $X = \langle x_1, x_2, \dots, x_m \rangle$, une séquence $Z = \langle z_1, z_2, \dots, z_k \rangle$ est une **sous-séquence** de X s'il existe une séquence $\langle i_1, i_2, \dots, i_k \rangle$ strictement croissante d'indices de X tels que, pour tout $j = 1, 2, \dots, k$, on ait $x_{i_j} = z_j$. Par exemple, $Z = \langle B, C, D, B \rangle$ est une sous-séquence de $X = \langle A, B, C, B, D, A, B \rangle$, correspondant à la séquence d'indices $\langle 2, 3, 5, 7 \rangle$.

Étant données deux séquences X et Y , on dit qu'une séquence Z est une **sous-séquence commune** de X et Y si Z est une sous-séquence de X et de Y . Par exemple, si $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$, la séquence $\langle B, C, A \rangle$ est une sous-séquence commune de X et de Y . Toutefois, la séquence $\langle B, C, A \rangle$ n'est pas une **plus longue** sous-séquence commune (PLSC) de X et Y , puisqu'elle est de longueur 3 et que la séquence $\langle B, C, B, A \rangle$, qui est aussi commune à X et Y , est de longueur 4. La séquence $\langle B, C, B, A \rangle$ est une PLSC de X et Y , de même que la séquence $\langle B, D, A, B \rangle$, puisqu'il n'existe pas de sous-séquence commune de longueur supérieure ou égale à 5.

Dans le **problème de la plus longue sous-séquence commune**, on dispose au départ de deux séquences $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$, et on souhaite trouver une sous-séquence commune à X et Y de longueur maximale. Cette section montrera que le problème de la PLSC peut être résolu efficacement grâce à la programmation dynamique.

a) Étape 1 : caractérisation d'une plus longue sous-séquence commune

La technique primaire de résolution du problème de la PLSC consiste à énumérer toutes les sous-séquences de X et les tester pour voir si elles sont aussi des sous-séquences de Y , en mémorisant au cours de la recherche la plus longue sous-séquence

trouvée. Chaque sous-séquence de X correspond à un sous-ensemble des indices $\{1, 2, \dots, m\}$ de X . Il existe 2^m sous-séquences de X , de sorte que cette approche demande un traitement à temps exponentiel, ce qui rend cette technique peu pratique pour les longues séquences.

Or, le problème de la PLSC possède une propriété de sous-structure optimale, comme le montre le théorème ci-après. Comme nous le verrons, les classes naturelles de sous-problèmes correspondent à des paires de « préfixes » des deux séquences d'entrée. Plus précisément, étant donnée une séquence $X = \langle x_1, x_2, \dots, x_m \rangle$, on définit le i ème **préfixe** de X , pour $i = 0, 1, \dots, m$, par $X_i = \langle x_1, x_2, \dots, x_i \rangle$. Par exemple, si $X = \langle A, B, C, B, D, A, B \rangle$, alors $X_4 = \langle A, B, C, B \rangle$ et X_0 représente la séquence vide.

Théorème 15.1 (Sous-structure optimale d'une PLSC) *Soient deux séquences $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$, et soit $Z = \langle z_1, z_2, \dots, z_k \rangle$ une PLSC quelconque de X et Y .*

- 1) Si $x_m = y_n$, alors $z_k = x_m = y_n$ et Z_{k-1} est une PLSC de X_{m-1} et Y_{n-1} .
- 2) Si $x_m \neq y_n$, alors $z_k \neq x_m$ implique que Z est une PLSC de X_{m-1} et Y .
- 3) Si $x_m \neq y_n$, alors $z_k \neq y_n$ implique que Z est une PLSC de X et Y_{n-1} .

Démonstration : (1) Si $z_k \neq x_m$, on peut concaténer $x_m = y_n$ à Z pour obtenir une sous-séquence commune de X et Y de longueur $k + 1$, ce qui contredit l'hypothèse selon laquelle Z est une *plus longue* sous-séquence commune de X et Y . On a donc forcément $z_k = x_m = y_n$. Or, le préfixe Z_{k-1} est une sous-séquence commune de longueur $(k - 1)$ de X_{m-1} et Y_{n-1} . On souhaite prouver que c'est une PLSC. Supposons, en raisonnant par l'absurde, qu'il existe une sous-séquence commune W de X_{m-1} et Y_{n-1} de longueur plus grande que $k - 1$. Alors, la concaténation de $x_m = y_n$ à W produit une sous-séquence commune à X et Y dont la longueur est plus grande que k , ce qui aboutit à une contradiction.

(2) Si $z_k \neq x_m$, alors Z est une sous-séquence commune de X_{m-1} et Y . S'il existait une sous-séquence commune W de X_{m-1} et Y de taille supérieure à k , alors W serait aussi une sous-séquence commune de X_m et Y , ce qui contredit l'hypothèse selon laquelle Z est une PLSC de X et Y .

(3) La démonstration est la symétrique de (2). □

La caractérisation du théorème 15.1 montre qu'une PLSC de deux séquences contient une PLSC de préfixes des deux séquences. Le problème de la PLSC possède donc une propriété de sous-structure optimale. Une solution récursive possède également la propriété de chevauchement des sous-problèmes, comme nous allons le voir bientôt.

b) Étape 2 : une solution récursive

Le théorème 15.1 implique que nous serons confrontés à un ou à deux sous-problèmes pendant la recherche d'une PLSC de $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$. Si $x_m = y_n$, on doit trouver une PLSC de X_{m-1} et Y_{n-1} . La concaténation de $x_m = y_n$ à cette PLSC engendre une PLSC de X et Y . Si $x_m \neq y_n$, alors on doit résoudre deux sous-problèmes : trouver une PLSC de X_{m-1} et Y , et trouver une PLSC de

X et Y_{n-1} . La plus grande des deux PLSC, quelle qu'elle soit, est une PLSC de X et Y . Comme ces cas épuisent toutes les possibilités, on sait que l'une des solutions optimales des sous-problèmes doit servir dans une PLSC de X et Y .

On peut déjà apercevoir la propriété de chevauchement des sous-problèmes dans le problème de la PLSC. Pour trouver une PLSC de X et Y , on peut avoir besoin de trouver les PLSC de X et Y_{n-1} et de X_{m-1} et Y . Mais chacun de ces sous-problèmes contient le sous-sous-problème consistant à trouver la PLSC de X_{m-1} et Y_{n-1} . Moutlt autres sous-problèmes ont des sous-sous-problèmes en commun.

Comme pour le problème de la multiplication d'une suite de matrices, notre solution récursive du problème de la PLSC implique la création d'une récurrence pour la valeur d'une solution optimale. Appelons $c[i, j]$ la longueur d'une PLSC des séquences X_i et Y_j . Si $i = 0$ ou $j = 0$, l'une des séquences a une longueur nulle, et donc la PLSC est de longueur nulle. La sous-structure optimale du problème de la PLSC débouche sur la formule récursive

$$c[i, j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ c[i - 1, j - 1] + 1 & \text{si } i, j > 0 \text{ et } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{si } i, j > 0 \text{ et } x_i \neq y_j. \end{cases} \quad (15.14)$$

Observez que, dans cette formulation récursive, il y a une condition du problème qui restreint la gamme des sous-problèmes à considérer. Quand $x_i = y_j$, on peut et doit considérer le sous-problème consistant à trouver la PLSC de X_{i-1} et Y_{j-1} . Autrement, on considère les deux sous-problèmes consistant à trouver la PLSC de X_i et Y_{j-1} et celle de X_{i-1} et Y_j . Dans nos précédents algorithmes de programmation dynamique (ordonnancement de chaîne de montage et multiplications de matrices), aucun sous-problème n'était éliminé en raison de conditions inhérentes au problème. Trouver la PLSC n'est pas le seul algorithme de programmation dynamique qui élimine des sous-problèmes en fonction de conditions posées dans le problème. Ainsi, le problème de la distance d'édition (voir problème 15.3) possède cette caractéristique.

c) Étape 3 : calcul de la longueur d'une PLSC

En se rapportant à l'équation (15.14), on pourrait facilement écrire un algorithme récursif à temps exponentiel pour calculer la longueur d'une PLSC de deux séquences. Cependant, comme il n'existe que $\Theta(mn)$ sous-problèmes distincts, on peut faire appel à la programmation dynamique pour calculer les solutions de manière ascendante.

La procédure LONGUEUR-PLSC prend deux séquences $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$ en entrée. Elle stocke les valeurs $c[i, j]$ dans un tableau $c[0..m, 0..n]$ dont les éléments sont calculés dans l'ordre des lignes. (Autrement dit, la première ligne de c est remplie de la gauche vers la droite, puis la deuxième ligne, etc.) Elle gère aussi un tableau $b[1..m, 1..n]$ pour simplifier la construction d'une solution optimale. Intuitivement, $b[i, j]$ pointe vers l'élément de tableau qui correspond à la solution optimale du sous-problème choisie pendant le calcul de $c[i, j]$. La procédure retourne les tableaux b et c ; $c[m, n]$ contient la longueur d'une PLSC de X et Y .

LONGUEUR-PLSC(X, Y)

```

1   $m \leftarrow \text{longueur}[X]$ 
2   $n \leftarrow \text{longueur}[Y]$ 
3  pour  $i \leftarrow 1$  à  $m$ 
4      faire  $c[i, 0] \leftarrow 0$ 
5  pour  $j \leftarrow 0$  à  $n$ 
6      faire  $c[0, j] \leftarrow 0$ 
7  pour  $i \leftarrow 1$  à  $m$ 
8      faire pour  $j \leftarrow 1$  à  $n$ 
9          faire si  $x_i = y_j$ 
10             alors  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
11                  $b[i, j] \leftarrow \llcorner \searrow \gg$ 
12             sinon si  $c[i - 1, j] \geq c[i, j - 1]$ 
13                 alors  $c[i, j] \leftarrow c[i - 1, j]$ 
14                      $b[i, j] \leftarrow \llcorner \uparrow \gg$ 
15                 sinon  $c[i, j] \leftarrow c[i, j - 1]$ 
16                      $b[i, j] \leftarrow \llcorner \leftarrow \gg$ 
17  retourner  $c$  et  $b$ 
    
```

La figure 15.6 montre les tableaux produits par LONGUEUR-PLSC sur les séquences $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$. Le temps d'exécution de la procédure est $O(mn)$, puisque chaque élément de tableau requiert un temps de calcul $O(1)$.

		j	0	1	2	3	4	5	6
		i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	←1	←1	↖1
2	B	0	↖1	↖1	↖1	1	↖2	←2	↖2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2	↖2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3	↖3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3	↖3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4	↑4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4	↖4

Figure 15.6 Les tableaux c et b calculés par LONGUEUR-PLSC sur les séquences $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$. Le carré à l'intersection de la ligne i et de la colonne j contient la valeur de $c[i, j]$ et la flèche appropriée pour la valeur de $b[i, j]$. La valeur 4 de $c[7, 6]$ (coin en bas à droite dans le tableau) est la longueur d'une PLSC $\langle B, C, B, A \rangle$ de X et Y . Pour $i, j > 0$, l'élément $c[i, j]$ ne dépend que de l'égalité éventuelle entre x_i et y_j et des valeurs des éléments $c[i - 1, j]$, $c[i, j - 1]$ et $c[i - 1, j - 1]$, qui sont calculés avant $c[i, j]$. Pour raccorder les éléments d'une PLSC, il suffit de suivre les flèches $b[i, j]$ à partir du coin inférieur droit ; le chemin est indiqué en gris. Chaque « \searrow » du chemin correspond à un élément (auréolé) pour lequel $x_i = y_j$ est un membre d'une PLSC.

d) Étape 4 : construction d'une PLSC

Le tableau b retourné par LONGUEUR-PLSC peut servir à construire rapidement une PLSC de $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$. On commence tout simplement en $b[m, n]$ et on se déplace dans le tableau en suivant les flèches. Chaque fois que nous rencontrons une « ↖ » dans l'élément $b[i, j]$, on sait que $x_i = y_j$ appartient à la PLSC. Cette méthode permet de retrouver les éléments de la PLSC dans l'ordre inverse. La procédure récursive suivante imprime une PLSC de X et Y dans le bon ordre. L'appel initial est IMPRIMER-PLSC($b, X, \text{longueur}[X], \text{longueur}[Y]$).

Pour le tableau b de la figure 15.6, cette procédure imprime « BCBA ». La procédure prend un temps $O(m + n)$, puisqu'au moins l'un des deux indices i ou j est décrémenté à chaque étape de la récursivité.

```

IMPRIMER-PLSC( $b, X, i, j$ )
1  si  $i = 0$  ou  $j = 0$ 
2    alors retourner
3  si  $b[i, j] = \text{« ↖ »}$ 
4    alors IMPRIMER-PLSC( $b, X, i - 1, j - 1$ )
5           imprimer  $x_i$ 
6  sinon si  $b[i, j] = \text{« ↑ »}$ 
7    alors IMPRIMER-PLSC( $b, X, i - 1, j$ )
8  sinon IMPRIMER-PLSC( $b, X, i, j - 1$ )

```

e) Amélioration du code

Une fois qu'un algorithme est développé, on s'aperçoit souvent qu'il est possible d'améliorer son coût en temps ou en espace. Cela est particulièrement vrai des algorithmes qui appliquent directement les préceptes de la programmation dynamique. Certaines modifications simplifient le code et améliorent les facteurs constants, mais ne vont pas jusqu'à améliorer asymptotiquement l'efficacité. D'autres peuvent apporter des économies asymptotiques substantielles en temps et en espace.

Par exemple, on peut éliminer complètement le tableau b . Chaque élément $c[i, j]$ ne dépend que de trois autres éléments du tableau c : $c[i - 1, j - 1]$, $c[i - 1, j]$, et $c[i, j - 1]$. Étant donnée la valeur de $c[i, j]$, on peut déterminer dans un temps $O(1)$ laquelle de ces trois valeurs a servi à calculer $c[i, j]$, sans parcourir le tableau b . On peut donc reconstruire une PLSC en temps $O(m + n)$ en utilisant une procédure similaire à IMPRIMER-PLSC. (L'exercice 15.4.2 vous demandera d'en donner le pseudo code.) Bien que nous ayons ainsi économisé $\Theta(mn)$ espace, l'espace auxiliaire requis pour calculer une PLSC ne décroît pas asymptotiquement, puisque nous avons de toute façon besoin d'un espace $\Theta(mn)$ pour le tableau c .

Néanmoins, il est possible de réduire les besoins asymptotiques en espace de LONGUEUR-PLSC, puisque cette procédure ne fait appel qu'à deux lignes du tableau c à un instant donné : la ligne en cours de calcul, et la précédente. (En fait, on

peut n'utiliser qu'un peu plus de l'espace nécessaire pour une lignes de c pour calculer la longueur d'une PLSC. Voir exercice 15.4.4.) Cette optimisation n'est possible que si l'on n'a besoin que de la longueur d'une PLSC ; si l'on doit reconstruire les éléments d'une PLSC, le tableau réduit ne contient plus assez d'informations pour retracer nos étapes en temps $O(m + n)$.

Exercices

15.4.1 Déterminer une PLSC de $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ et $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

15.4.2 Montrer comment reconstruire une PLSC à partir du tableau c complet et des séquences initiales $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$ dans un temps $O(m + n)$, sans utiliser le tableau b .

15.4.3 Donner une version à recensement de LONGUEUR-PLSC qui s'exécute en temps $O(mn)$.

15.4.4 Montrer comment calculer la longueur d'une PLSC en n'ayant recours qu'à $2 \cdot \min(m, n)$ éléments du tableau c , plus un espace supplémentaire $O(1)$. Ensuite, montrer comment faire la même chose en utilisant $\min(m, n)$ éléments plus un espace supplémentaire $O(1)$.

15.4.5 Donner un algorithme à temps $O(n^2)$ pour trouver la plus longue sous-séquence monotone croissante d'une séquence de n nombres.

15.4.6 * Donner un algorithme à temps $O(n \lg n)$ pour trouver la plus longue sous-séquence monotone croissante d'une séquence de n nombres. (*conseil* : Remarquer que le dernier élément d'une sous-séquence idoine de longueur i est au moins aussi grand que le dernier élément d'une sous-séquence idoine de longueur $i - 1$. Mémoriser les sous-séquences idoines en les reliant à l'intérieur de la séquence d'entrée.)

15.5 ARBRES BINAIRES DE RECHERCHE OPTIMAUX

Supposez que l'on veuille écrire un programme qui traduise des textes de l'anglais vers le français. Pour chaque occurrence de chaque mot anglais, il faut chercher l'équivalent français. Une façon d'implémenter ces recherches consiste à créer un arbre binaire de recherche contenant n mots anglais comme clés, les équivalents français étant les données satellite. Comme on explorera l'arbre pour chaque mot du texte, on veut minimiser le temps total de consultation. On pourrait garantir un temps de recherche $O(\lg n)$ par occurrence en employant un arbre rouge-noir, ou toute autre espèce d'arbre binaire de recherche équilibré. Mais les mots ont des fréquences d'apparition différentes ; or, il peut advenir qu'un mot très courant tel que « the » soit loin de la racine et qu'un mot rare comme « mycophagist » soit près de la racine. Une telle disposition ralentit la traduction, vu que le nombre de nœuds visités lors de la

recherche d'une clé dans un arbre binaire vaut un de plus que la profondeur du nœud contenant la clé. On veut que les mots qui reviennent souvent soient près de la racine. (3) En outre, il peut exister des mots anglais pour lesquels il n'existe aucune traduction en français ; ces mots n'apparaissent donc pas du tout dans l'arbre. Comment organiser un arbre binaire de recherche de façon à minimiser le nombre de nœuds visités dans toutes les recherches, connaissant la fréquence d'apparition de chaque mot ?

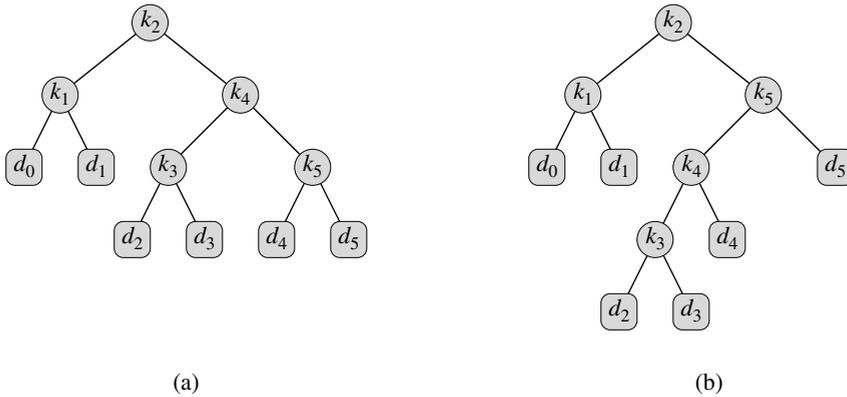


Figure 15.7 Deux arbres binaires de recherche pour un ensemble de $n = 5$ clés ayant les probabilités suivantes :

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) Un arbre binaire de recherche ayant un coût de recherche moyen de 2,80.

(b) Un arbre binaire de recherche ayant un coût de recherche moyen de 2,75. Cet arbre est optimal.

Ce qu'il nous faut, c'est un **arbre binaire de recherche optimal**. Plus formellement, soit une séquence $K = \langle k_1, k_2, \dots, k_n \rangle$ de n clés distinctes triées (de sorte que $k_1 < k_2 < \dots < k_n$) ; on veut construire un arbre binaire de recherche optimal à partir de ces clés. Pour chaque clé k_i , on a la probabilité p_i qu'une recherche concerne k_i . Certaines recherches pouvant concerner des valeurs n'appartenant pas à K , on a donc aussi $n + 1$ « clés factices » $d_0, d_1, d_2, \dots, d_n$ représentant des valeurs extérieures à K . En particulier, d_0 représente toutes les valeurs inférieures à k_1 , d_n représente toutes les valeurs supérieures à k_n et, pour $i = 1, 2, \dots, n - 1$, la clé factice d_i représente toutes les valeurs comprises entre k_i et k_{i+1} . Pour chaque clé factice d_i , on a une probabilité q_i qu'une recherche concerne d_i . La figure 15.7 montre deux arbres binaires de recherche pour un ensemble de $n = 5$ clés. Chaque clé k_i est un nœud interne, et chaque clé factice d_i est une feuille. Chaque recherche soit réussit (elle trouve une

(3) Si le texte traite des champignons comestibles, on pourrait vouloir que « mycophagiste » soit près de la racine.

certaine clé k_i), soit échoue (elle trouve une certaine clé factice d_i), et donc on a

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1 . \quad (15.15)$$

Comme on a des probabilités de recherche pour chaque clé et pour chaque clé factice, on peut déterminer le coût moyen d'une recherche dans un arbre binaire donné T . Supposons que le coût réel d'une recherche soit le nombre de nœuds examinés, c'est-à-dire la profondeur du nœud trouvé par la recherche plus 1. Alors, le coût moyen d'une recherche dans T est : $E[\text{coût de recherche dans } T]$

$$\begin{aligned} &= \sum_{i=1}^n (\text{profondeur}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{profondeur}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{profondeur}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{profondeur}_T(d_i) \cdot q_i , \end{aligned} \quad (15.16)$$

où profondeur_T désigne la profondeur d'un nœud dans T . La dernière égalité découle de l'équation (15.15). Sur la figure 15.7(a), on peut calculer le coût de recherche moyen nœud par nœud :

nœud	profondeur	probabilité	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Pour un ensemble donné de probabilités, on veut construire un arbre binaire de recherche qui minimise le coût de recherche moyen. À propos d'un tel arbre, on parle d'**arbre binaire de recherche optimal**. La figure 15.7(b) montre un arbre binaire de recherche optimal pour les probabilités données dans la figure ; son coût attendu est 2,75. Cet exemple montre qu'un arbre binaire de recherche optimal n'est pas forcément un arbre dont la hauteur globale est minimale. L'exemple montre aussi que l'on ne construit pas forcément un arbre binaire de recherche optimal en plaçant systématiquement à la racine la clé ayant la plus grande probabilité. Ici, c'est la clé k_5 qui a la probabilité de recherche maximale, et pourtant la racine de l'arbre binaire de recherche optimal contient k_2 . (Le coût moyen minimal d'un arbre binaire de recherche ayant k_5 comme racine est 2,85.)

Comme c'est le cas avec la multiplication de matrices en chaîne, la vérification exhaustive de toutes les possibilités ne donne pas un algorithme efficace. On peut étiqueter les nœuds d'un arbre binaire à n nœuds avec les clés k_1, k_2, \dots, k_n pour construire un arbre binaire de recherche, puis ajouter les clés factices comme feuilles. Au problème 12.4, on a vu que le nombre d'arbres binaires à n nœuds est $\Omega(4^n/n^{3/2})$; il y aurait donc un nombre exponentiel d'arbres binaires à examiner si l'on adoptait la méthode primitive. Vous ne serez pas étonné d'apprendre que nous allons résoudre ce problème à l'aide de la programmation dynamique.

a) Étape 1 : structure d'un arbre binaire de recherche optimal

Pour caractériser la sous-structure optimale des arbres binaire de recherche optimaux, commençons par une observation sur les sous-arbres. Soit un sous-arbre d'un arbre binaire de recherche. Il doit contenir des clés appartenant à une plage contiguë k_i, \dots, k_j , pour un certain $1 \leq i \leq j \leq n$. En outre, un sous-arbre qui contient les clés k_i, \dots, k_j doit avoir comme feuilles les clés factices d_{i-1}, \dots, d_j .

Nous pouvons maintenant énoncer la sous-structure optimale : si un arbre binaire de recherche optimal T a un sous-arbre T' contenant les clés k_i, \dots, k_j , alors ce sous-arbre T' est optimal pour le sous-problème ayant les clés k_i, \dots, k_j et les clés factices d_{i-1}, \dots, d_j . L'argument couper-coller usuel s'applique : s'il y avait un sous-arbre T'' dont le coût moyen est inférieur à celui de T' , alors on pourrait couper T' dans T puis coller T'' , ce qui donnerait un arbre binaire de recherche dont le coût est inférieur à celui de T , ce qui contredirait l'optimalité de T .

Nous devons utiliser la sous-structure optimale pour montrer que l'on peut construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. Soient les clés k_i, \dots, k_j ; l'une de ces clés, par exemple k_r ($i \leq r \leq j$), est la racine d'un sous-arbre optimal contenant ces clés. Le sous-arbre gauche de la racine k_r contiendra les clés k_i, \dots, k_{r-1} (et les clés factices d_{i-1}, \dots, d_{r-1}); le sous-arbre droite contiendra les clés k_{r+1}, \dots, k_j (et les clés factices d_r, \dots, d_j). Si l'on examine toutes les candidats k_r pour la place de racine (avec $i \leq r \leq j$) et si l'on détermine tous les arbres binaire de recherche optimaux contenant k_i, \dots, k_{r-1} et ceux contenant k_{r+1}, \dots, k_j , alors on est certain de trouver un arbre binaire de recherche optimal.

Il y a un détail qui vaut la peine d'être noté, concernant les sous-arbres « vides ». Supposez que, dans un sous-arbre de clés k_i, \dots, k_j , on sélectionne k_i comme racine. De par le raisonnement précédent, le sous-arbre gauche de k_i contient les clés k_i, \dots, k_{i-1} . Il est naturel d'interpréter cette séquence comme ne contenant aucune clé. N'oubliez pas, toutefois, que les sous-arbres contiennent aussi des clés factices. Nous adopterons la convention suivante : un sous-arbre contenant les clés k_i, \dots, k_{i-1} n'a pas de clés véritables, mais il contient la clé factice d_{i-1} . De manière symétrique, si l'on prend k_j comme racine, alors le sous-arbre droite de k_j contient les clés k_{j+1}, \dots, k_j ; ce sous-arbre ne contient pas de clés, mais contient la clé factice d_j .

b) *Étape 2 : une solution récursive*

Nous voici parés pour définir récursivement la valeur d'une solution optimale. Définissons notre domaine de sous-problème : trouver un arbre binaire de recherche optimal contenant les clés k_i, \dots, k_j , avec $i \geq 1$, $j \leq n$ et $j \geq i - 1$. (C'est quand $j = i - 1$ qu'il n'y a pas de clés, mais juste la clé factice d_{i-1} .) Soit $e[i, j]$ le coût moyen de recherche dans un arbre binaire de recherche optimal contenant les clés k_i, \dots, k_j . Le but final est de calculer $e[1, n]$.

Le cas facile se produit pour $j = i - 1$. On a alors uniquement la clé factice d_{i-1} . Le coût de recherche moyen est $e[i, i - 1] = q_{i-1}$.

Quand $j \geq i$, on doit sélectionner une racine k_r parmi k_i, \dots, k_j , puis faire un arbre binaire de recherche optimal contenant les clés k_i, \dots, k_{r-1} de son sous-arbre gauche et faire un arbre binaire de recherche optimal contenant les clés k_{r+1}, \dots, k_j de son sous-arbre droite. Qu'advient-il du coût de recherche moyen d'un sous-arbre quand il devient un sous-arbre d'un nœud ? La profondeur de chaque nœud du sous-arbre augmente de 1. D'après l'équation (15.16), le coût de recherche moyen de ce sous-arbre augmente de la somme de toutes les probabilités du sous-arbre. Pour un sous-arbre de clés k_i, \dots, k_j , notons ainsi cette somme de probabilités

$$w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l. \quad (15.17)$$

Donc, si k_r est la racine d'un sous-arbre optimal contenant les clés k_i, \dots, k_j , on a

$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)).$$

En remarquant que

$$w(i, j) = w(i, r - 1) + p_r + w(r + 1, j),$$

on réécrit $e[i, j]$ ainsi

$$e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j). \quad (15.18)$$

L'équation récursive (15.18) suppose que nous sachions quel est le nœud k_r à prendre comme racine. On choisit la racine qui donne le coût de recherche moyen le plus faible ; d'où la formulation récursive finale :

$$e[i, j] = \begin{cases} q_{i-1} & \text{si } j = i - 1, \\ \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w(i, j)\} & \text{si } i \leq j. \end{cases} \quad (15.19)$$

Les valeurs $e[i, j]$ donnent les coûts de recherche moyens dans des arbres binaires de recherche optimaux. Pour nous aider à gérer la structure des arbres binaires de recherche optimaux, définissons *racine* $[i, j]$, pour $1 \leq i \leq j \leq n$, comme étant l'indice r pour lequel k_r est la racine d'un arbre binaire de recherche optimal contenant les clés k_i, \dots, k_j . Nous verrons comment calculer les valeurs de *racine* $[i, j]$, mais nous laisserons à l'exercice 15.5.1 le soin de construire l'arbre binaire de recherche optimal à partir de ces valeurs.

c) *Étape 3 : calcul du coût de recherche moyen dans un arbre binaire de recherche optimal*

À ce stade, vous avez peut-être noté certaines similitudes entre les caractérisations des arbres binaires de recherche optimaux et des multiplications matricielles en chaîne. Pour ces deux classes de problème, les sous-problèmes se composent de sous-ensembles d'indices contigus. Une implémentation récursive directe de l'équation (15.19) serait aussi inefficace qu'un algorithme récursif direct de multiplications matricielles en chaîne. À la place, nous allons stocker les valeurs $e[i, j]$ dans un tableau $e[1 \dots n + 1, 0 \dots n]$. Le premier indice doit aller jusqu'à $n + 1$ et non n car, pour avoir un sous-arbre ne contenant que la clé factice d_n , on doit calculer et stocker $e[n + 1, n]$. Le second indice doit partir de 0 car, pour avoir un sous-arbre contenant uniquement la clé factice d_0 , on doit calculer et stocker $e[1, 0]$. On n'utilisera que les éléments $e[i, j]$ pour lesquels $j \geq i - 1$. On emploiera aussi un tableau *racine* $[i, j]$, pour mémoriser la racine du sous-arbre contenant les clés k_i, \dots, k_j . Ce tableau n'utilise que les éléments pour lesquels $1 \leq i \leq j \leq n$.

On aura besoin d'un autre tableau, à des fins d'efficacité. Au lieu de calculer la valeur de $w(i, j)$ ex nihilo chaque fois que l'on calcule $e[i, j]$ (ce qui prendrait $\Theta(j - i)$ additions), on stocke ces valeurs dans un tableau $w[1 \dots n + 1, 0 \dots n]$. Pour le cas de base, on calcule $w[i, i - 1] = q_{i-1}$ pour $1 \leq i \leq n$. Pour $j \geq i$, on calcule

$$w[i, j] = w[i, j - 1] + p_j + q_j. \quad (15.20)$$

On peut donc calculer les $\Theta(n^2)$ valeurs de $w[i, j]$ avec un temps $\Theta(1)$ pour chacune. Le pseudo code qui suit prend en entrée les probabilités p_1, \dots, p_n et q_0, \dots, q_n et la taille n , puis retourne les tableaux e et *racine*.

ABR-OPTIMAL(p, q, n)

```

1  pour  $i \leftarrow 1$  à  $n + 1$ 
2      faire  $e[i, i - 1] \leftarrow q_{i-1}$ 
3           $w[i, i - 1] \leftarrow q_{i-1}$ 
4  pour  $l \leftarrow 1$  à  $n$ 
5      faire pour  $i \leftarrow 1$  à  $n - l + 1$ 
6          faire  $j \leftarrow i + l - 1$ 
7               $e[i, j] \leftarrow \infty$ 
8               $w[i, j] \leftarrow w[i, j - 1] + p_j + q_j$ 
9              pour  $r \leftarrow i$  à  $j$ 
10                 faire  $t \leftarrow e[i, r - 1] + e[r + 1, j] + w[i, j]$ 
11                 si  $t < e[i, j]$ 
12                     alors  $e[i, j] \leftarrow t$ 
13                          $racine[i, j] \leftarrow r$ 
14  retourner  $e$  et racine
```

Compte tenu de la description précédente et de la similitude avec la procédure ORDRE-CHAÎNE-MATRICES de la section 15.2, vous devriez comprendre sans trop de problèmes

le fonctionnement de cette procédure. La boucle **pour** des lignes 1–3 initialise les valeurs de $e[i, i - 1]$ et $w[i, i - 1]$. La boucle **pour** des lignes 4–13 utilise ensuite les récurrences (15.19) et (15.20) pour calculer $e[i, j]$ et $w[i, j]$ pour tout $1 \leq i \leq j \leq n$. Dans la première itération, quand $l = 1$, la boucle calcule $e[i, i]$ et $w[i, i]$ pour $i = 1, 2, \dots, n$. La deuxième itération, avec $l = 2$, calcule $e[i, i + 1]$ et $w[i, i + 1]$ pour $i = 1, 2, \dots, n - 1$, etc. La boucle **pour** la plus interne, en lignes 9–13, essaie chaque indice candidat r pour déterminer quelle est la clé k_r à utiliser comme racine d'un arbre binaire de recherche optimal contenant les clés k_i, \dots, k_j . Cette boucle **pour** mémorise la valeur courante de l'indice r dans $racine[i, j]$ chaque fois qu'elle trouve une clé meilleure pour servir de racine.

La figure 15.8 montre les tables $e[i, j]$, $w[i, j]$ et $racine[i, j]$ calculées par la procédure ABR-OPTIMAL pour la distribution de clés de la figure 15.7. Comme dans l'exemple des produits de matrices en chaîne, on a fait tourner les tableaux pour rendre horizontales les diagonales. ABR-OPTIMAL calcule les lignes du bas vers le haut, puis de gauche à droite dans chaque ligne.

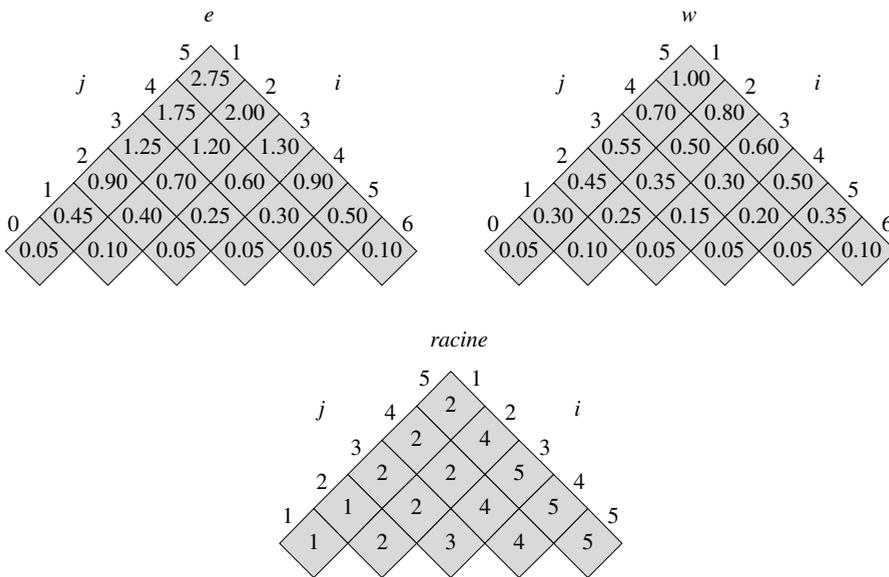


Figure 15.8 Tableaux $e[i, j]$, $w[i, j]$ et $racine[i, j]$ calculés par ABR-OPTIMAL sur la distribution de clés de la figure 15.7. Les tableaux sont inclinés de façon que les diagonales soient à l'horizontale.

ABR-OPTIMAL prend un temps $\Theta(n^3)$, tout comme ORDRE-CHAÎNE-MATRICES. Il est facile de voir que le temps d'exécution est $O(n^3)$, car les boucles **pour** sont imbriquées sur trois niveaux et chaque indice de boucle prend au plus n valeurs. Les indices de boucle de ABR-OPTIMAL n'ont pas exactement les mêmes limites que ceux de ORDRE-CHAÎNE-MATRICES, mais la différence est de 1 au plus dans toutes les directions. Donc, comme ORDRE-CHAÎNE-MATRICES, la procédure ABR-OPTIMAL prend un temps $\Omega(n^3)$.

Exercices

15.5.1 Écrire le pseudo code de la procédure CONSTRUIRE-ABR-OPTIMAL(*racine*) qui, à partir de la table *racine*, affiche la structure d'un arbre binaire de recherche optimal. Pour l'exemple de la figure 15.8, votre procédure devra afficher la structure

k_2 est la racine
 k_1 est l'enfant gauche de k_2
 d_0 est l'enfant gauche de k_1
 d_1 est l'enfant droite de k_1
 k_5 est l'enfant droite de k_2
 k_4 est l'enfant gauche de k_5
 k_3 est l'enfant gauche de k_4
 d_2 est l'enfant gauche de k_3
 d_3 est l'enfant droite de k_3
 d_4 est l'enfant droite de k_4
 d_5 est l'enfant droite de k_5

correspondant à l'arbre binaire de recherche optimal montré sur la figure 15.7(b).

15.5.2 Déterminer le coût et la structure d'un arbre binaire de recherche optimal pour un ensemble de $n = 7$ clés ayant les probabilités suivantes :

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

15.5.3 Supposez que, au lieu de gérer le tableau $w[i, j]$, on calcule la valeur de $w(i, j)$ directement à partir de l'équation (15.17) en ligne 8 de ABR-OPTIMAL puis que l'on utilise à la ligne 10 cette valeur calculée. Comment cette modification affecterait-elle le temps d'exécution asymptotique de ABR-OPTIMAL ?

15.5.4 * Knuth [184] a montré qu'il y a toujours des racines de sous-arbres optimaux telles que $racine[i, j - 1] \leq racine[i, j] \leq racine[i + 1, j]$ pour tout $1 \leq i < j \leq n$. Utiliser ce fait pour modifier ABR-OPTIMAL pour qu'elle tourne en temps $\Theta(n^2)$.

PROBLÈMES

15.1. Problème euclidien bitonique du voyageur de commerce

Le *problème du voyageur de commerce euclidien* consiste à déterminer la plus petite tournée permettant de relier un ensemble donné de n points du plan. La figure 15.9(a) montre la solution pour un problème à 7 points. Le problème général est NP-complet, et il y a donc de fortes chances pour que sa solution demande un temps suprapolynomial (voir chapitre 34).

J. L. Bentley a suggéré de simplifier le problème en se restreignant aux *tournées bitoniques*, autrement dit, celles qui partent du point le plus à gauche, continuent strictement de gauche à droite vers le point le plus à droite, puis retournent vers le point de départ en se déplaçant strictement de droite à gauche. La figure 15.9(b) montre la plus courte tournée bitonique pour ces mêmes 7 points. Dans ce cas, on peut trouver un algorithme à temps polynomial.

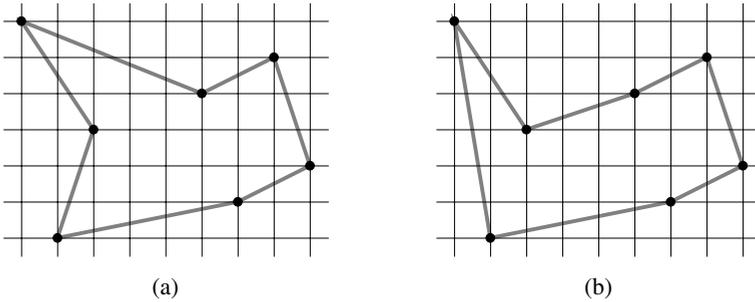


Figure 15.9 Sept points du plan, représentés sur une grille orthonormée. (a) La tournée la plus courte, d’une longueur de 24,88... Cette tournée n’est pas bitonique. (b) La tournée bitonique la plus courte pour le même ensemble de points. Sa longueur fait environ 25,58...

Décrire un algorithme à temps $O(n^2)$ pour déterminer une tournée bitonique optimale. Vous pouvez supposer que deux points n’ont jamais la même abscisse. (*conseil* : Balayer le plan de gauche à droite, en mémorisant des possibilités optimales pour les deux parties de la tournée.)

15.2. Impression équilibrée

On considère le problème d’une impression équilibrée d’un paragraphe sur une imprimante. Le texte d’entrée est une séquence de n mots de longueurs l_1, l_2, \dots, l_n , mesurées en caractères. On souhaite imprimer ce paragraphe de façon équilibrée sur un certain nombre de lignes qui contiennent un maximum de M caractères chacune. Notre critère d’« équilibre » est le suivant. Si une ligne donnée contient les mots i à j , où $i \leq j$, et qu’on laisse exactement un espace entre deux mots, le nombre de caractères d’espacement supplémentaires à la fin de la ligne est $M - j + i - \sum_{k=i}^j l_k$, qui doit être positif ou nul, pour que les mots tiennent sur la ligne. On souhaite minimiser la somme, sur toutes les lignes hormis la dernière, des cubes des nombres de caractères d’espacement présents à la fin de chaque ligne. Donner un algorithme de programmation dynamique permettant d’imprimer de manière équilibrée un paragraphe de n mots sur une imprimante. Analyser les besoins en temps et en espace de votre algorithme.

15.3. Distance d’édition

Pour transformer une chaîne textuelle source $x[1..m]$ en une chaîne cible $y[1..n]$, on peut effectuer diverses transformations. Notre but est le suivant : étant donné x

et y , on veut produire une série de transformations qui changent x en y . On utilise un tableau z (censé être suffisamment grand pour contenir tous les caractères nécessaires) pour stocker les résultats intermédiaires. Initialement z est vide, et à la fin on doit avoir $z[j] = y[j]$ pour $j = 1, 2, \dots, n$. On gère les indices courants i pour x et j pour z ; les opérations ont le droit de modifier z et ces indices. Initialement, $i = j = 1$. Comme on est obligé d'examiner chaque caractère de x pendant la transformation, à la fin de la séquence de transformations on doit avoir $i = m + 1$.

Il y a six transformations possibles :

Copier: un caractère de x vers z en faisant $z[j] \leftarrow x[i]$, puis en incrémentant i et j . Cette opération examine $x[i]$.

Remplacer: un caractère de x par un autre caractère c en faisant $z[j] \leftarrow c$, puis en incrémentant i et j . Cette opération examine $x[i]$.

Supprimer: un caractère de x en incrémentant i mais en ne touchant pas à j . Cette opération examine $x[i]$.

Insérer: le caractère c dans z en faisant $z[j] \leftarrow c$, puis en incrémentant j sans toucher à i . Cette opération n'examine aucun caractère de x .

Permuter: les deux caractères suivants, en les copiant de x vers z mais dans l'ordre inverse ; pour ce faire, on fait $z[j] \leftarrow x[i+1]$ et $z[j+1] \leftarrow x[i]$ puis l'on fait $i \leftarrow i+2$ et $j \leftarrow j+2$. Cette opération examine $x[i]$ et $x[i+1]$.

Équeuter: le reste de x en faisant $i \leftarrow m + 1$. Cette opération examine tous les caractères de x qui n'ont pas encore été examinés. Si l'on fait cette opération, ce doit être la dernière de la série.

À titre d'exemple, une façon de transformer la chaîne source `algorithm` en la chaîne cible `altruistic` est d'employer la séquence suivante d'opérations ; les caractères soulignés représentent $x[i]$ et $z[j]$ après l'opération :

Opération	x	z
<i>chaînes initiales</i>	<u>a</u> lgorithm	_
copier	a <u>l</u> gorithm	a_
copier	al <u>g</u> orithm	al_
remplacer par t	alg <u>o</u> rithm	alt_
supprimer	algor <u>u</u> ithm	alt_
copier	algori <u>t</u> h	altr_
insérer u	algori <u>u</u> th	altru_
insérer i	algori <u>i</u> th	altrui_
insérer s	algori <u>s</u> th	altru <u>i</u> s_
permuter	algori <u>u</u> th	altru <u>i</u> st <u>i</u> _
insérer c	algori <u>u</u> th	altru <u>i</u> st <u>i</u> c_
équeuter	algori <u>u</u> th_	altru <u>i</u> st <u>i</u> c_

Notez que ce n'est pas la seule séquence de transformations qui change `algorithm` en `altruistic`.

Chacune des opérations a un coût. Le coût d'une opération dépend de l'application spécifique, mais on supposera ici que le coût de chaque opération est une constante qui est connue. On supposera aussi que les coûts individuels des opérations copier et remplacer sont inférieurs aux coûts combinés des opérations supprimer et insérer ; sinon, on n'utiliserait pas les opérations copier et remplacer. Le coût d'une séquence de transformations est la somme des coûts des opérations de la séquence. Pour la séquence précédente, le coût de transformation de `algorithm` en `altruistic` est

$$(3 \cdot \text{coût}(\text{copier})) + \text{coût}(\text{remplacer}) + \text{coût}(\text{supprimer}) + (4 \cdot \text{coût}(\text{insérer})) \\ + \text{cot}(\text{permuter}) + \text{coût}(\text{équeuter}) .$$

- a. Étant données deux séquences $x[1 \dots m]$ et $y[1 \dots n]$ et un ensemble de coûts d'opération, la **distance d'édition** de x à y est le coût de la séquence d'opérations la plus économique qui transforme x en y . Donner un algorithme de programmation dynamique qui détermine la distance d'édition de $x[1 \dots m]$ à $y[1 \dots n]$ et affiche une séquence d'opérations optimale. Analyser les besoins en temps et en espace de cet algorithme.

Le problème de la distance d'édition est une généralisation du problème de l'alignement de deux séquences ADN (voir, par exemple, Setubal et Meidanis [272, Section 3.2]). Il existe plusieurs méthodes pour mesurer la similitude de deux séquences ADN en les alignant. L'une de ces méthodes d'alignement de deux séquences x et y consiste à insérer des espaces à des emplacements arbitraires (extrémités comprises) dans les deux séquences, de façon que les séquences résultantes x' et y' aient la même longueur mais n'aient pas un espace au même emplacement (quel que soit j , on n'a jamais un espace dans $x'[j]$ et $y'[j]$ à la fois.) Ensuite, on assigne une « pondération » à chaque emplacement. La position j reçoit un poids de la façon suivante :

- +1 si $x'[j] = y'[j]$ et aucun des deux n'est un espace,
- -1 si $x'[j] \neq y'[j]$ et aucun des deux n'est un espace,
- -2 si l'un de $x'[j]$ ou $y'[j]$ est un espace.

Le poids de l'alignement est la somme des poids des emplacements. Ainsi, étant données les séquences $x = \text{GATCGGCAT}$ et $y = \text{CAATGTGAATC}$, un alignement possible est

```
G ATCG GCAT
CAAT GTGAATC
- * + + * + * + - + + *
```

Un + sous un emplacement indique une pondération de +1, un - indique un poids de -1 et un * indique un poids de -2 ; cet alignement a donc un poids totale de $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

- b. Expliquer comment transformer le problème de l'alignement optimal en un problème de distance d'édition utilisant un sous-ensemble des transformations copier, remplacer, supprimer, insérer, permuter et équeuter.

15.4. Planification d'un raout d'entreprise

Le professeur Knock est consultant pour le compte du PDG d'une entreprise qui veut organiser un raout d'entreprise. L'entreprise a une structure hiérarchisée arborescente, dont le PDG est la racine. Le service du personnel a affecté à chaque employé une note de convivialité, qui est un nombre réel. Pour que le raout soit une vraie fête pour tout le monde, le PDG ne veut pas inviter en même temps un employé et son supérieur direct.

Le professeur Knock se voit remettre l'arborescence qui décrit la structure de l'entreprise à l'aide de la représentation enfant-gauche, frère-droite vue à la section 10.4. Chaque nœud de l'arborescence contient, outre les pointeurs, le nom d'un employé et sa note de convivialité. Décrire un algorithme pour établir une liste d'invités qui maximise le total des notes de convivialité des invités. Analyser le temps d'exécution de cet algorithme.

15.5. Algorithme de Viterbi

On peut se servir de la programmation dynamique sur un graphe orienté $G = (S, A)$ pour la reconnaissance vocale. Chaque arc $(u, v) \in A$ est étiqueté par un phonème $\sigma(u, v)$ tiré d'un ensemble fini Σ de phonèmes. Le graphe ainsi étiqueté modélise une personne parlant un langage restreint. Chaque chemin du graphe partant d'un sommet distingué $v_0 \in S$ correspond à une séquence possible de phonèmes produite par le modèle. L'étiquette d'un chemin orienté est définie comme étant la concaténation des étiquettes des arcs du chemin.

- a. Décrire un algorithme efficace qui, étant donné un graphe G à arcs étiquetés et sommet distingué v_0 et une séquence $s = \langle \sigma_1, \sigma_2, \dots, \sigma_k \rangle$ de caractères de Σ , retourne un chemin de G qui commence à v_0 et a l'étiquette s , si un tel chemin existe. Sinon, l'algorithme retournera CHEMIN-INEXISTANT. Analyser le temps d'exécution de votre algorithme. (*Conseil* : Les concepts abordés au chapitre 22 pourront vous être utiles.)

Supposons maintenant qu'on attribue aussi à chaque arc $(u, v) \in A$ une probabilité positive ou nulle $p(u, v)$, représentant les chances de passer par l'arc (u, v) à partir du sommet u et donc de produire le son correspondant. La somme des probabilités des arcs partant d'un sommet quelconque est égale à 1. La probabilité d'un chemin est définie comme étant le produit des probabilités de ses arcs. On peut voir la probabilité d'un chemin commençant en v_0 comme étant la probabilité pour qu'une « marche aléatoire » commençant en v_0 suive le chemin en question, le choix de l'arc à prendre à partir d'un sommet u se faisant selon les probabilités des arcs disponibles depuis u .

- b. Étendre la réponse de la question (a) de façon que le chemin retourné soit un *chemin le plus probable* partant de v_0 et ayant l'étiquette s . Analyser le temps d'exécution de votre algorithme.

15.6. Déplacement sur un damier

Supposez que vous ayez un damier $n \times n$ et un jeton. Vous devez déplacer le jeton depuis le bord inférieur du damier vers le bord supérieur, en respectant la règle suivante. À chaque étape, vous pouvez placer le jeton sur l'un des trois carrés suivants :

- 1) le carré qui est juste au dessus,
- 2) le carré qui est situé une position plus haut et une position plus à gauche (à condition que le jeton ne soit pas déjà dans la colonne la plus à gauche),
- 3) le carré qui est situé une position plus haut et une position plus à droite (à condition que le jeton ne soit pas déjà dans la colonne la plus à droite).

Chaque fois que vous passez du carré x au carré y , vous recevez $p(x, y)$ euros. On vous donne $p(x, y)$ pour toutes les paires (x, y) correspondant à un déplacement licite de x à y . $p(x, y)$ n'est pas forcément positif.

Donner un algorithme pour déterminer l'ensemble des déplacements qui feront passer le jeton du bord inférieur au bord supérieur, tout en vous faisant empocher le maximum d'euros. Votre algorithme peut partir de n'importe quel carré du bord inférieur et arriver sur n'importe quel carré du bord supérieur pour maximiser le montant collecté au cours du trajet. Quel est le temps d'exécution ?

15.7. Ordonnement à profit maximal

Vous avez une machine et n tâches a_1, a_2, \dots, a_n à traiter sur cette machine. Chaque tâche a_j a une durée d'exécution t_j , un profit p_j et une date d'échéance d_j . La machine ne peut traiter qu'une seule tâche à la fois, et la tâche a_j doit s'exécuter sans interruption pendant t_j unités de temps consécutives. Si la tâche a_j est terminée avant la limite d_j , vous recevez un gain p_j ; sinon, vous ne recevez rien. Donner un algorithme pour déterminer l'ordonnement permettant d'obtenir le profit maximal, en supposant que tous les temps de traitement soient des entiers compris entre 1 et n . Quel est le temps d'exécution ?

NOTES

R. Bellman commença l'étude systématique de la programmation dynamique en 1955. Le mot « programmation », dans ce contexte comme dans celui de la programmation linéaire, fait référence à l'utilisation d'une méthode de résolution tabulaire. Bien que des techniques d'optimisation incorporant des éléments de programmation dynamique fussent connues depuis longtemps, Bellman les justifia par une solide base mathématique. [34].

Hu et Shing [159, 160] donnent un algorithme à temps $O(n \lg n)$ pour le problème de la multiplication d'une suite de matrices.

L'algorithme à temps $O(mn)$ donné pour le problème de la plus longue sous-séquence commune semble être un algorithme utilisé depuis longtemps. Knuth [63] pose la question de savoir si des algorithmes sous-quadratiques existaient pour le problème de la PLSC. Masek

et Paterson [212] répondirent à cette question par l'affirmative, en donnant un algorithme qui s'exécute en temps $O(mn/\lg n)$, où $n \leq m$ et où les séquences sont tirées d'un ensemble de taille bornée. Pour le cas particulier où aucun élément n'apparaît plus d'une fois dans la séquence d'entrée, Szymanski [288] montre que le problème peut être résolu en temps $O((n+m)\lg(n+m))$. Beaucoup de ces résultats s'étendent au problème du calcul de distances d'édition de chaînes. (problème 15.3).

Un premier article sur les encodages binaires de longueur variable, dû à Gilbert et Moore [114], expliquait comment construire des arbres binaires de recherche optimaux pour le cas où toutes les probabilités p_i sont 0 ; cet article contenait un algorithme à temps $O(n^3)$. Aho, Hopcroft et Ullman [5] présentent l'algorithme de la section 15.5. L'exercice 15.5.4 est dû à Knuth [184]. Hu et Tucker [161] ont conçu un algorithme pour le cas où toutes les probabilités p_i sont 0, qui utilise $O(n^2)$ temps et $O(n)$ espace ; par la suite, Knuth [185] a réduit le temps à $O(n \lg n)$.

Chapitre 16

Algorithmes gloutons

Les algorithmes pour problèmes d'optimisation exécutent en général une série d'étapes, chaque étape proposant un ensemble de choix. Pour de nombreux problèmes d'optimisation, la programmation dynamique est une approche bien trop lourde pour déterminer les meilleurs choix ; d'autres algorithmes, plus simples et plus efficaces, peuvent faire l'affaire. Un *algorithme glouton* fait toujours le choix qui lui semble le meilleur sur le moment. Autrement dit, il fait un choix localement optimal dans l'espoir que ce choix mènera à une solution globalement optimale. Ce chapitre étudie les problèmes d'optimisation qui peuvent se résoudre par des algorithmes gloutons. Avant de lire ce chapitre, mieux vaut avoir lu le chapitre 15 consacré à la programmation dynamique, et notamment la section 15.3.

Les algorithmes gloutons n'aboutissent pas toujours à des solutions optimales, mais ils y arrivent dans de nombreux cas. Nous commencerons par examiner à la section 16.1 un problème simple mais non trivial, à savoir le problème du choix d'activités, pour lequel un algorithme glouton calcule une solution efficacement. Nous arriverons à l'algorithme glouton en commençant par étudier une solution basée sur la programmation dynamique, puis en montrant que l'on peut toujours faire des choix gloutons pour arriver à une solution optimale. La section 16.2 passera en revue les éléments fondamentaux de l'approche gloutonne et donnera une méthode plus directe, pour prouver la validité des algorithmes gloutons, que le processus de la section 16.1 basé sur la programmation dynamique. La section 16.3 présentera une application importante des techniques gloutonnes : la conception de codes (de Huffman) de compression de données. Dans la section 16.4, nous nous étudierons des structures combinatoires appelées « matroïdes » pour lesquelles un algorithme glouton produit

toujours une solution optimale. Enfin, la section 16.5 illustrera l'utilité des matroïdes sur le problème de l'ordonnancement temporel de tâches, avec dates d'échéance et pénalités.

La méthode gloutonne est très puissante et fonctionne bien pour toutes sortes de problèmes. Des chapitres ultérieurs présenteront de nombreux algorithmes qui peuvent être vus comme des applications de la méthode gloutonne, notamment les algorithmes d'arbre couvrant minimal (chapitre 23), l'algorithme de Dijkstra qui calcule des plus courts chemins à origine unique (chapitre 24) et l'heuristique gloutonne de Chvátal pour le recouvrement d'ensemble (chapitre 35). Les algorithmes d'arbre couvrant minimal sont un exemple classique de la méthode gloutonne. Bien que ce chapitre et le chapitre 23 puissent être lus indépendamment, il peut être utile de les lire ensemble.

16.1 UN PROBLÈME DE CHOIX D'ACTIVITÉS

Notre premier exemple concernera le problème de l'ordonnancement de plusieurs activités qui rivalisent pour l'utilisation exclusive d'une ressource commune, l'objectif étant de sélectionner un ensemble de taille maximale d'activités mutuellement compatibles. Supposez que l'on ait un ensemble $S = \{a_1, a_2, \dots, a_n\}$ de n **activités** proposées qui veulent utiliser une ressource, par exemple une salle de conférences qui ne peut servir qu'à une seule activité à la fois. Chaque activité a_i a une **heure de début** s_i et une **heure de fin** f_i , avec $0 \leq s_i < f_i < \infty$. Si elle est sélectionnée, l'activité a_i a lieu pendant l'intervalle temporel semi-ouvert $[s_i, f_i)$. Les activités a_i et a_j sont **compatibles** si les intervalles $[s_i, f_i)$ et $[s_j, f_j)$ ne se chevauchent pas (c'est-à-dire que a_i et a_j sont compatibles si $s_i \geq f_j$ ou $s_j \geq f_i$). Le **problème du choix d'activités** consiste à sélectionner un sous-ensemble, de taille maximale, d'activités mutuellement compatibles. Par exemple, considérons l'ensemble suivant S d'activités, que nous avons trié par ordre croissant d'heure de fin :

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(Nous verrons bientôt l'intérêt de traiter les activités dans cet ordre.) Pour cet exemple, le sous-ensemble $\{a_3, a_9, a_{11}\}$ se compose d'activités mutuellement compatibles. Ce n'est pas un sous-ensemble maximum, cependant, car le sous-ensemble $\{a_1, a_4, a_8, a_{11}\}$ est plus grand. En fait, $\{a_1, a_4, a_8, a_{11}\}$ est un sous-ensemble maximum d'activités mutuellement compatibles ; un autre sous-ensemble maximum est $\{a_2, a_4, a_9, a_{11}\}$.

Nous résoudrons ce problème en plusieurs étapes. Nous commencerons par formuler une solution basée sur la programmation dynamique, dans laquelle on combine les solutions optimales de deux sous-problèmes pour former une solution optimale du problème initial. On envisagera plusieurs choix quand on déterminera quels sont les

sous-problèmes à utiliser dans une solution optimale. On observera alors qu'il suffit de considérer un seul choix, à savoir le choix glouton, et que, quand on opte pour le choix glouton, on a la certitude que l'un des sous-problèmes sera vide et qu'il ne restera donc qu'un seul sous-problème. En partant de ces observations, nous développerons un algorithme glouton récursif pour résoudre le problème de l'ordonnement d'activités. Nous terminerons le processus de développement d'une solution gloutonne en convertissant l'algorithme récursif en algorithme itératif. Bien que les étapes par lesquelles nous passerons dans cette section soient plus complexes que ce qui se fait généralement pour le développement d'un algorithme glouton, elles illustrent bien les relations qui existent entre algorithmes gloutons et programmation dynamique.

a) La sous-structure optimale du problème du choix d'activités

Comme précédemment mentionné, nous commencerons par développer une solution basée sur la programmation dynamique pour le problème du choix d'activités. Comme dans le chapitre 15, notre première étape sera de trouver la sous-structure optimale puis de nous en servir pour construire une solution optimale du problème à partir des solutions optimales de sous-problèmes.

Nous avons vu au chapitre 15 que nous avons besoin de définir un espace approprié de sous-problèmes. Commençons par définir les ensembles

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} ,$$

tels que S_{ij} soit le sous-ensemble d'activités de S qui peuvent démarrer après que l'activité a_i s'est terminée et finir avant que l'activité a_j démarre. En fait, S_{ij} se compose de toutes les activités qui sont compatibles avec a_i et a_j , et qui sont aussi compatibles avec toutes les activités qui ne finissent pas plus tard que a_i et avec toutes les activités qui ne commencent pas plus tôt que a_j . Pour représenter l'ensemble du problème, nous ajouterons des activités fictives a_0 et a_{n+1} et adopterons les conventions selon lesquelles $f_0 = 0$ et $s_{n+1} = \infty$. Alors, $S = S_{0,n+1}$ et les intervalles de i et j sont donnés par $0 \leq i, j \leq n + 1$.

Nous pouvons restreindre encore les intervalles de i et j , de la manière suivante. Supposons que les activités soient triées par ordre monotone croissant d'heure de fin :

$$f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1} . \quad (16.1)$$

Nous affirmons que $S_{ij} = \emptyset$ chaque fois que $i \geq j$. Pourquoi ? Supposons qu'il existe une activité $a_k \in S_{ij}$ pour un certain couple $i \geq j$, de sorte que a_i suit a_j dans notre ordre. On aurait alors $f_i \leq s_k < f_k \leq s_j < f_j$. Donc, on aurait $f_i < f_j$, ce qui contredit notre hypothèse que a_i suit a_j dans notre ordre. On en conclut que, si l'on a trié les activités par ordre monotone croissant d'heure de fin, notre espace de sous-problèmes consiste à sélectionner un sous-ensemble, de taille maximale, d'activités mutuellement compatibles prises dans S_{ij} , pour $0 \leq i < j \leq n + 1$, sachant que tous les autres S_{ij} sont vides.

Pour voir la sous-structure du problème de choix d'activités, considérons un certain sous-problème non vide S_{ij} ,⁽¹⁾ et supposons qu'une solution à S_{ij} contienne une certaine activité a_k , de sorte que $f_i \leq s_k < f_k \leq s_j$. Utiliser l'activité a_k génère deux sous-problèmes, S_{ik} (activités qui démarrent après que a_i a fini et qui finissent avant que a_k démarre) et S_{kj} (activités qui démarrent après que a_k a fini et qui finissent avant que a_j démarre), dont chacun se compose d'un sous-ensemble des activités de S_{ij} . Notre solution à S_{ij} est l'union des solutions à S_{ik} et des solutions à S_{kj} , plus l'activité a_k . Donc, le nombre d'activités de notre solution à S_{ij} est égale à la taille de notre solution à S_{ik} , plus la taille de notre solution à S_{kj} , plus un (pour a_k).

La sous-structure optimale de ce problème est la suivante. Supposons maintenant qu'une solution optimale A_{ij} de S_{ij} contienne l'activité a_k . Alors, les solutions A_{ik} de S_{ik} et A_{kj} de S_{kj} utilisées dans cette solution optimale de S_{ij} doivent être, elles aussi, optimales. On peut appliquer le raisonnement « couper-coller » habituel. Si l'on avait une solution A'_{ik} de S_{ik} qui contienne plus d'activités que A_{ik} , on couperait A_{ik} de A_{ij} et on le collerait dans A'_{ik} , ce qui produirait une autre solution de S_{ij} contenant plus d'activités que A_{ij} . Comme on a supposé que A_{ij} est une solution optimale, on aboutit à une contradiction. De même, si l'on avait une solution A'_{kj} de S_{kj} ayant plus d'activités que A_{kj} , on pourrait remplacer A_{kj} par A'_{kj} pour produire une solution de S_{ij} ayant plus d'activités que A_{ij} .

Nous allons maintenant utiliser notre sous-structure optimale pour montrer que nous pouvons construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. Nous avons vu que toute solution d'un sous-problème non vide S_{ij} contient une certaine activité a_k , et que toute solution optimale contient en elle des solutions optimales des instances de sous-problème S_{ik} et S_{kj} . Donc, on peut construire un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_{ij} en divisant le problème en deux sous-problèmes (trouver des sous-ensembles, de taille maximale, d'activités mutuellement compatibles de S_{ik} et S_{kj}), en trouvant des sous-ensembles de taille maximale A_{ik} et A_{kj} d'activités mutuellement compatibles pour ces sous-problèmes, puis en formant notre sous-ensemble de taille maximale A_{ij} d'activités mutuellement compatibles de la façon suivante

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} . \quad (16.2)$$

Une solution optimale pour le problème tout entier est une solution de $S_{0,n+1}$.

b) Une solution récursive

La deuxième phase de la création d'une solution basée sur la programmation dynamique consiste à définir récursivement la valeur d'une solution optimale. Pour le problème du choix d'activités, soit $c[i, j]$ le nombre d'activités d'un sous-ensemble de

(1) Nous parlerons parfois, à propos des ensembles S_{ij} , de sous-problèmes au lieu d'ensembles d'activités. Le contexte indiquera toujours si l'on se réfère à S_{ij} en tant qu'ensemble d'activités, ou en tant que sous-problème dont l'entrée est cet ensemble.

taille maximale d'activités mutuellement compatibles de S_{ij} . On a $c[i, j] = 0$ chaque fois que $S_{ij} = \emptyset$; en particulier, $c[i, j] = 0$ pour $i \geq j$.

Considérons maintenant un sous-ensemble non vide S_{ij} . Nous avons vu que, si a_k appartient à un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_{ij} , on utilise aussi des sous-ensembles de taille maximale d'activités mutuellement compatibles pour les sous-problèmes S_{ik} et S_{kj} . En utilisant l'équation (16.2), on obtient la récurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

Cette équation récursive suppose que l'on connaisse la valeur de k , que nous ne connaissons pas. Il y a $j - i - 1$ valeurs possibles pour k , à savoir $k = i + 1, \dots, j - 1$. Comme le sous-ensemble de taille maximale de S_{ij} doit utiliser l'une de ces valeurs pour k , on les teste toutes afin de trouver la meilleure. Notre définition entièrement récursive de $c[i, j]$ devient donc

$$c[i, j] = \begin{cases} 0 & \text{si } S_{ij} = \emptyset , \\ \max_{i < k < j \text{ et } a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{si } S_{ij} \neq \emptyset . \end{cases} \quad (16.3)$$

c) Conversion d'une solution de type programmation dynamique en une solution gloutonne

À ce stade, il serait on ne peut plus facile d'écrire un algorithme de programmation dynamique, tabulaire et ascendant (bottom-up), basé sur la récurrence (16.3). En fait, ce sera justement l'objet de l'exercice 16.11. Il y a, toutefois, deux observations majeures qui vont nous permettre de simplifier notre solution.

Théorème 16.1 *Soit un sous-problème non vide S_{ij} et soit a_m l'activité de S_{ij} ayant l'heure de fin la plus précoce :*

$$f_m = \min \{f_k : a_k \in S_{ij}\} .$$

Alors

- 1) L'activité a_m est utilisée dans un certain sous-ensemble de taille maximale d'activités mutuellement compatibles de S_{ij} .
- 2) Le sous-problème S_{im} est vide, de sorte que choisir a_m fait du sous-problème S_{mj} le seul sous-problème susceptible d'être non vide.

Démonstration : Nous allons commencer par démontrer la seconde partie, vu qu'elle est un peu plus simple. Supposons que S_{im} soit non vide, auquel cas il existe une activité a_k telle que $f_i \leq s_k < f_k \leq s_m < f_m$. Alors, a_k est aussi dans S_{ij} et elle a une heure de fin antérieure à celle de a_m , ce qui contredit notre choix de a_m . On en conclut que S_{im} est vide.

Pour prouver la première partie, supposons que A_{ij} soit un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_{ij} , et rangeons les activités de A_{ij} par ordre monotone croissant d'heure de fin. Soit a_k la première activité de A_{ij} .

Si $a_k = a_m$, on a fini, car on a montré que a_m est utilisée dans un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_{ij} . Si $a_k \neq a_m$, construisons le sous-ensemble $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$. Les activités de A'_{ij} sont disjointes, vu que les activités de A_{ij} le sont, a_k est la première activité de A_{ij} à finir et $f_m \leq f_k$. En remarquant que A'_{ij} a le même nombre d'activités que A_{ij} , on voit que A'_{ij} est un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_{ij} qui contient a_m . \square

Pourquoi le théorème 16.1 est-il si précieux ? Rappelez-vous (voir section 15.3) que la sous-structure optimale dépend du nombre de sous-problèmes qui sont utilisés dans une solution optimale du problème original, ainsi que du nombre de choix que nous avons pour déterminer quels sont les sous-problèmes à utiliser. Dans notre solution de type programmation dynamique, il y a deux sous-problèmes qui sont utilisés dans une solution optimale et il y a $j - i - 1$ choix quand on résout le sous-problème S_{ij} . Le théorème 16.1 réduit sensiblement ces deux quantités : il ne reste plus qu'un sous-problème dans une solution optimale (l'autre étant forcément vide) et, quand on résout le sous-problème S_{ij} , on n'a plus qu'un seul choix à considérer, à savoir celui qui a l'heure de fin la plus précoce dans S_{ij} . Fort heureusement, il est facile de déterminer de quelle activité il s'agit.

Outre la réduction du nombre de sous-problèmes et du nombre de choix, le théorème 16.1 procure un autre avantage : on peut résoudre chaque sous-problème de manière descendante (top-down), et non de manière ascendante comme c'est généralement le cas en programmation dynamique. Pour résoudre le sous-problème S_{ij} , on choisit l'activité a_m de S_{ij} qui a l'heure de fin la plus précoce et l'on ajoute à cette solution l'ensemble des activités utilisées dans une solution optimale du sous-problème S_{mj} . Comme on sait que, ayant choisi a_m , on va certainement utiliser une solution de S_{mj} dans notre solution optimale de S_{ij} , on n'a pas besoin de résoudre S_{mj} avant de résoudre S_{ij} . Pour résoudre S_{ij} , on peut d'abord choisir a_m comme étant l'activité de S_{ij} qui a l'heure de fin la plus précoce et ensuite résoudre S_{mj} .

Notez aussi qu'il y a une loi générale sous-jacente aux sous-problèmes que nous résolvons. Notre problème originel est $S = S_{0,n+1}$. Supposons que nous choisissons a_{m_1} comme étant l'activité de $S_{0,n+1}$ qui a l'heure de fin la plus précoce. (Comme nous avons trié les activités par ordre monotone croissant d'heure de fin et que $f_0 = 0$, on a forcément $m_1 = 1$.) Notre sous-problème suivant est $S_{m_1,n+1}$. Supposons maintenant que nous choisissons a_{m_2} comme étant l'activité de $S_{m_1,n+1}$ qui a l'heure de fin la plus précoce. (On n'a pas obligatoirement $m_2 = 2$.) Notre sous-problème suivant est $S_{m_2,n+1}$. En continuant ainsi, on voit que chaque sous-problème est de la forme $S_{m_i,n+1}$ où m_i désigne un certain numéro d'activité. Autrement dit, chaque sous-problème se compose des dernières activités à finir et le nombre de ces activités varie d'un sous-problème à l'autre.

Il y a aussi une loi générale sous-jacente aux activités que nous choisissons. Comme nous choisissons systématiquement l'activité qui a l'heure de fin la plus précoce dans $S_{m_i,n+1}$, les heures de fin des activités choisies, tous sous-problèmes

confondus, croissent strictement avec le temps. En outre, on peut traiter chaque activité une seule fois, dans l'ordre monotone croissant des heures de fin.

L'activité a_m que nous choisissons quand nous résolvons un sous-problème est toujours celle qui a l'heure de fin la plus précoce susceptible d'être planifiée en toute légalité. L'activité sélectionnée est donc un choix « glouton », au sens où intuitivement, elle laisse le maximum de possibilités pour les activités restant à planifier. Autrement dit, le choix glouton est celui qui maximise la quantité restante de temps non planifié.

d) Un algorithme glouton récursif

Maintenant que nous avons vu comment raffiner notre solution basée sur la programmation dynamique et comment la traiter comme une méthode descendante (top-down), nous sommes prêts pour voir un algorithme descendant qui fonctionne de manière 100 % gloutonne. Nous donnerons une solution récursive directe, sous la forme de la procédure CHOIX-D'ACTIVITÉS-RÉCURSIF. Elle prend en entrée les heures de début et de fin des activités, représentées par les tableaux s et f , ainsi que les indices i et n définissant le sous-problème $S_{i,n+1}$ qu'elle doit résoudre. (Le paramètre n indexe la dernière activité réelle a_n du sous-problème, et non l'activité fictive a_{n+1} qui appartient elle aussi au sous-problème.) Elle retourne un ensemble, de taille maximale, d'activités mutuellement compatibles de $S_{i,n+1}$. On suppose que les n activités données en entrée sont triées par ordre monotone croissant d'heures de fin, selon l'équation (16.1). Si tel n'est pas le cas, on peut les trier en temps $O(n \lg n)$ en rompant les égalités arbitrairement. L'appel initial est CHOIX-D'ACTIVITÉS-RÉCURSIF($s, f, 0, n$).

CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, i, n)

- 1 $m \leftarrow i + 1$
- 2 **tant que** $m \leq n$ et $s_m < f_i$ ▷ trouver première activité de $S_{i,n+1}$.
- 3 **faire** $m \leftarrow m + 1$
- 4 **si** $m \leq n$
- 5 **alors retourner** $\{a_m\} \cup$ CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, m, n)
- 6 **sinon retourner** \emptyset

La figure 16.1 montre le fonctionnement de l'algorithme. Dans un appel récursif donné CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, i, n), la boucle **tant que** des lignes 2–3 recherche la première activité de $S_{i,n+1}$. La boucle examine $a_{i+1}, a_{i+2}, \dots, a_n$, jusqu'à ce qu'elle trouve la première activité a_m qui est compatible avec a_i ; une telle activité a $s_m \geq f_i$. Si la boucle se termine parce qu'elle trouve une telle activité, la procédure retourne en ligne 5 l'union de $\{a_m\}$ et du sous-ensemble de taille maximale de $S_{m,n+1}$ retourné par l'appel récursif CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, m, n). La boucle peut aussi se terminer parce que $m \geq n$, auquel cas on a examiné toutes les activités sans en trouver une qui soit compatible avec a_i . Dans ce cas, $S_{i,n+1} = \emptyset$, et donc la procédure retourne \emptyset en ligne 6.

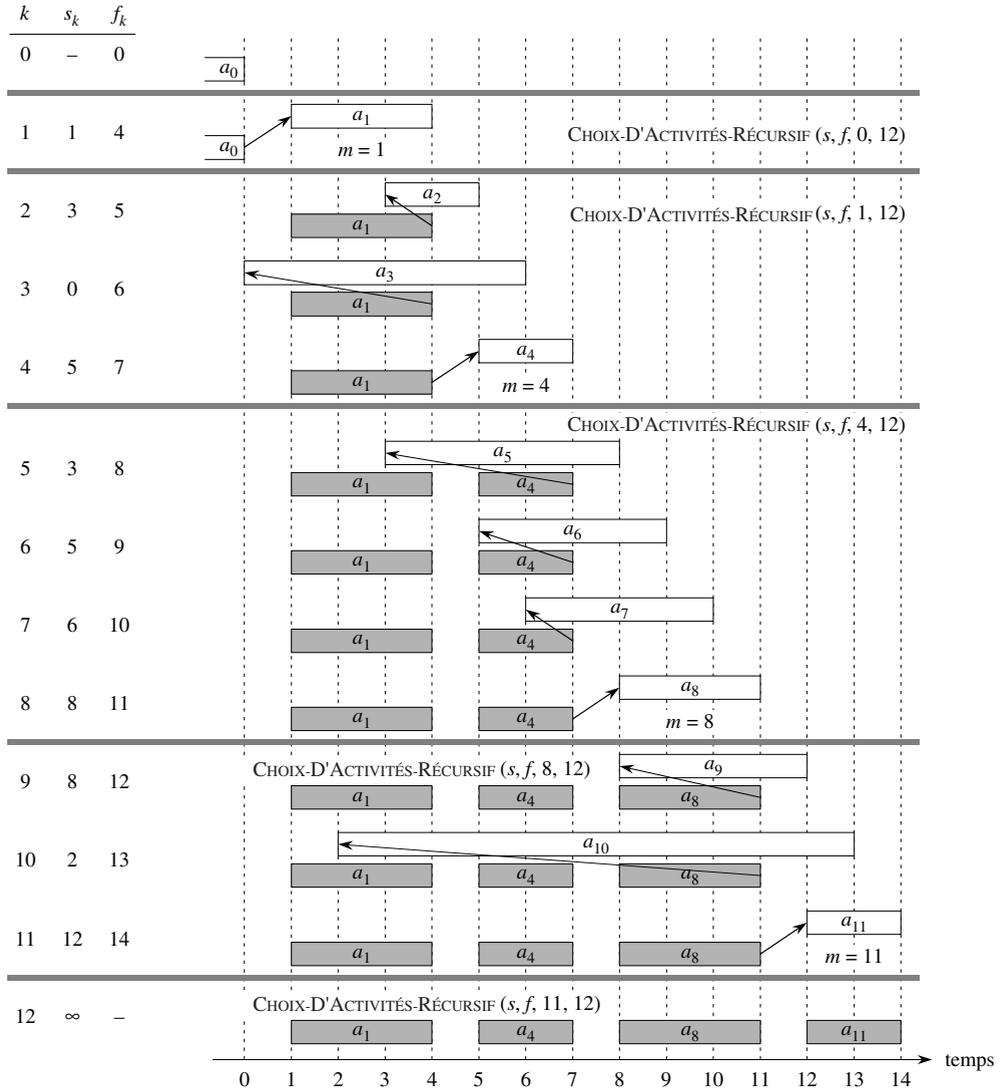


Figure 16.1 Fonctionnement de CHOIX-D'ACTIVITÉS-RÉCURSIF sur les 11 activités précédemment données. Les activités traitées dans chaque appel récursif apparaissent entre des traits horizontaux. L'activité fictive a_0 finit à l'heure 0 et, dans l'appel initial CHOIX-D'ACTIVITÉS-RÉCURSIF($s, f, 0, 11$), l'activité a_1 est sélectionnée. Dans chaque appel récursif, les activités qui ont été déjà sélectionnées sont estompées, et l'activité en blanc est celle qui est en cours de traitement. Si l'heure de début d'une activité a lieu avant l'heure de fin de l'activité ajoutée en dernier (la flèche entre elles pointe vers la gauche), elle est refusée. Sinon (la flèche pointe directement vers le haut ou vers la droite), elle est sélectionnée. Le dernier appel récursif, CHOIX-D'ACTIVITÉS-RÉCURSIF($s, f, 11, 11$), retourne \emptyset . L'ensemble d'activités sélectionnées résultant est $\{a_1, a_4, a_8, a_{11}\}$.

En supposant que les activités ont déjà été triées par heures de fin, la durée d'exécution de l'appel CHOIX-D'ACTIVITÉS-RÉCURSIF($s, f, 0, n$) est $\Theta(n)$, ce que l'on peut justifier comme suit. Sur l'ensemble des appels récursifs, chaque activité est examinée une fois et une seule dans le test de la boucle **tant que** en ligne 2. En particulier, l'activité a_k est examinée dans le dernier appel effectué dans lequel on a $i < k$.

e) Un algorithme glouton itératif

Il est facile de convertir notre procédure récursive en procédure itérative. La procédure CHOIX-D'ACTIVITÉS-RÉCURSIF est presque « récursive terminale » (voir problème 7.4) : elle se termine par un appel récursif à elle-même, suivi d'une opération d'union. C'est généralement un travail immédiat que de transformer une Procédure récursive terminale en une variante itérative ; d'ailleurs, certains compilateurs le font d'eux-mêmes. Telle qu'elle est écrite, CHOIX-D'ACTIVITÉS-RÉCURSIF fonctionne pour les sous-problèmes $S_{i,n+1}$, c'est-à-dire pour les sous-problèmes qui sont constitués des dernières activités à finir.

La procédure CHOIX-D'ACTIVITÉS-GLOUTON est une version itérative de CHOIX-D'ACTIVITÉS-RÉCURSIF. Elle suppose, elle aussi, que les activités données en entrée sont triées par ordre monotone croissant d'heures de fin. Elle recueille les activités sélectionnées dans un ensemble A , et elle retourne cet ensemble quand elle a fini.

CHOIX-D'ACTIVITÉS-GLOUTON(s, f)

```

1   $n \leftarrow \text{longueur}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  pour  $m \leftarrow 2$  à  $n$ 
5      faire si  $s_m \geq f_i$ 
6          alors  $A \leftarrow A \cup \{a_m\}$ 
7               $i \leftarrow m$ 
8  retourner  $A$ 
```

La procédure fonctionne de la manière suivante. La variable i indexe le tout dernier ajout à A , qui correspond à l'activité a_i dans la version récursive. Comme les activités sont traitées par ordre croissant d'heures de fin, f_i est toujours l'heure de fin maximale d'une quelconque activité de A . C'est-à-dire que

$$f_i = \max \{f_k : a_k \in A\} . \quad (16.4)$$

Les lignes 2–3 sélectionnent l'activité a_1 , initialisent A pour qu'il contienne uniquement cette activité, puis initialisent i pour indexer cette activité. La boucle **pour** des lignes 4–7 trouve l'activité qui finit le plus tôt dans $S_{i,n+1}$. La boucle traite chaque activité a_m à tour de rôle et ajoute a_m à A si elle est compatible avec toutes les activités déjà sélectionnées ; une telle activité est la première à finir dans $S_{i,n+1}$. Pour s'assurer

que l'activité a_m est compatible avec chaque activité actuellement dans A , il suffit d'après l'équation (16.4) de vérifier (ligne 5) que son heure de début s_m n'est pas antérieure à l'heure de fin f_i de l'activité qui a été ajoutée à A en dernier. Si l'activité a_m est compatible, alors les lignes 6–7 ajoutent a_m à A et affectent à i la valeur m . L'ensemble A retourné par l'appel CHOIX-D'ACTIVITÉS-GLOUTON(s, f) n'est autre que l'ensemble retourné par l'appel CHOIX-D'ACTIVITÉS-RÉCURSIF($s, f, 0, n$).

À l'instar de la version récursive, CHOIX-D'ACTIVITÉS-GLOUTON ordonnance un ensemble de n activités en un temps $\Theta(n)$, à condition que les activités aient été préalablement triés par heures de fin.

Exercices

16.1.1 Donner un algorithme de programmation dynamique pour le problème du choix d'activités, basé sur la récurrence (16.3). L'algorithme devra calculer les tailles $c[i, j]$ telles qu'elles ont été définies en amont et produire le sous-ensemble de taille maximale A d'activités. On supposera que les entrées ont été triées comme dans l'équation (16.1). Comparer le temps d'exécution de votre solution à celui de CHOIX-D'ACTIVITÉS-GLOUTON.

16.1.2 Supposez que, au lieu de sélectionner systématiquement la première activité à se terminer, on sélectionne la dernière activité à démarrer qui soit compatible avec toutes les activités précédemment sélectionnées. Expliquer en quoi cette approche est un algorithme glouton et prouver qu'elle donne une solution optimale.

16.1.3 Supposons qu'on ait un ensemble d'activités à répartir sur un grand nombre de salles de cours. On souhaite planifier toutes les activités avec le minimum de salles de cours. Donner un algorithme glouton efficace qui détermine quelle est l'activité qui doit avoir lieu dans telle ou telle salle de cours. (Ce problème est aussi connu sous le nom de *coloration d'un graphe d'intervalles*. On peut créer un graphe d'intervalles dont les sommets sont les activités données et dont les arêtes relient les activités incompatibles. Trouver le nombre minimal de couleurs pour colorier chaque sommet de telle manière que deux sommets adjacents n'aient jamais la même couleur, cela correspond à trouver le nombre minimal de salles nécessaires pour ordonnancer toutes les activités données.)

16.1.4 N'importe quelle approche gloutonne du problème du choix d'activités ne produit pas toujours un ensemble de taille maximale d'activités mutuellement compatibles. Donner un exemple montrant que l'approche qui consiste à choisir l'activité ayant la durée plus courte, parmi celles qui sont compatibles avec les activités déjà sélectionnées, ne fonctionne pas. Faire de même pour les approches suivantes : sélection systématique de l'activité compatible qui chevauche le moins possible d'autres activités restantes ; sélection systématique de l'activité restante compatible qui a l'heure de début la plus précoce.

16.2 ÉLÉMENTS DE LA STRATÉGIE GLOUTONNE

Un algorithme glouton détermine une solution optimale pour un problème après avoir effectué une série de choix. Pour chaque point de décision de l'algorithme, le choix

qui semble le meilleur à l'instant est effectué. Cette stratégie heuristique ne produit pas toujours une solution optimale mais, comme nous l'avons vu dans le problème du choix d'activités, c'est parfois le cas. Cette section étudie les propriétés générales des méthodes gloutonnes.

Le processus que nous avons suivi à la section 16.1, pour mettre au point un algorithme glouton, était un peu plus compliqué que ce qui se fait généralement. Nous sommes passés par les étapes suivantes :

- 1) Détermination de la sous-structure optimale du problème.
- 2) Développement d'une solution récursive.
- 3) Démonstration que, à chaque étape de la récursivité, l'un des choix optimaux est le choix glouton. Par conséquent, c'est toujours une décision correcte que de faire le choix glouton.
- 4) Démonstration que tous les sous-problèmes qui résultent du choix glouton, sauf un, sont vides.
- 5) Mise au point d'un algorithme récursif qui implémente la stratégie gloutonne.
- 6) Conversion de l'algorithme récursif en algorithme itératif.

En passant par ces étapes, nous avons vu de manière très détaillée les éléments de programmation dynamique sous-jacents à un algorithme glouton. En pratique, on allège les étapes susmentionnées quand on conçoit un algorithme glouton. On développe la sous-structure en ayant à l'esprit qu'il faudra faire un choix glouton qui ne laissera qu'un seul sous-problème à résoudre de manière optimale. Par exemple, dans le problème du choix d'activités, on a commencé par définir les sous-problèmes S_{ij} , avec i et j variant tous les deux. On a ensuite observé que, si l'on faisait toujours le choix glouton, on pouvait obliger les sous-problèmes à être de la forme $S_{i,n+1}$.

Une autre possibilité aurait consisté à dessiner la sous-structure optimale en ayant en tête un choix glouton. Autrement dit : on aurait pu laisser tomber le second indice et définir des sous-problèmes de la forme $S_i = \{a_k \in S : f_i \leq s_k\}$. Ensuite, on aurait prouvé qu'un choix glouton (la première activité a_m à finir dans S_i), combiné avec une solution optimale de l'ensemble S_m d'activités compatibles restant, donne une solution optimale de S_i . Plus généralement, on conçoit les algorithmes gloutons selon les étapes suivantes :

- 1) Transformation du problème d'optimisation en un problème dans lequel on fait un choix à la suite duquel on se retrouve avec un seul sous-problème à résoudre.
- 2) Démonstration qu'il y a toujours une solution optimale du problème initial qui fait le choix glouton, de sorte que le choix glouton est toujours approprié.
- 3) Démonstration que, après avoir fait le choix glouton, on se retrouve avec un sous-problème tel que, si l'on combine une solution optimale du sous-problème et le choix glouton que l'on a fait, on arrive à une solution optimale du problème original.

Nous emploierons ce processus plus direct dans les sections suivantes de ce chapitre. Il faut savoir néanmoins que, derrière chaque algorithme glouton, se cache presque toujours une solution plus lourde à base de programmation dynamique.

Comment peut-on savoir si un algorithme glouton saura résoudre un problème d'optimisation particulier ? C'est en général impossible, mais la propriété du choix glouton et la sous-structure optimale sont les deux éléments clé. Si l'on peut montrer que le problème possède ces propriétés, alors on est en bonne voie pour créer un algorithme glouton pour ce problème.

a) *Propriété du choix glouton*

Le première caractéristique principale est la **propriété du choix glouton** : on peut arriver à une solution globalement optimale en effectuant un choix localement optimal (glouton). Autrement dit : quand on considère le choix à faire, on fait le choix qui paraît le meilleur pour le problème courant, sans tenir compte des résultats des sous-problèmes.

C'est en cela que les algorithmes gloutons diffèrent de la programmation dynamique. En programmation dynamique, on fait un choix à chaque étape, mais ce choix dépend généralement de la solution des sous-problèmes. Par conséquent, on résout généralement les problèmes de programmation dynamique d'une manière ascendante, en partant de petits sous-problèmes pour arriver à des sous-problèmes plus gros. Dans un algorithme glouton, on fait le choix qui semble le meilleur sur le moment, puis on résout le sous-problème qui survient une fois que le choix est fait. Le choix effectué par un algorithme glouton peut dépendre des choix effectués jusque là, mais ne peut pas dépendre d'un quelconque choix futur ni des solutions des sous-problèmes. Ainsi, contrairement à la programmation dynamique, qui résout les sous-problèmes de manière ascendante, une stratégie gloutonne progresse en général de manière descendante, en faisant se succéder les choix gloutons, pour ramener itérativement chaque instance du problème à une instance plus petite.

Bien entendu, il faut démontrer qu'un choix glouton à chaque étape engendre une solution optimale globalement, et c'est là qu'un peu d'astuce peut s'avérer utile. La plupart du temps, comme dans le cas du théorème 16.1, la démonstration étudie une solution globalement optimale d'un certain sous-problème. Elle montre ensuite que la solution peut être modifiée pour utiliser le choix glouton, ce qui donnera un problème similaire mais de taille plus petite.

La propriété du choix glouton nous procurera souvent une certaine efficacité pour ce qui est de faire le choix dans un sous-problème. Par exemple, dans le problème du choix d'activités, si l'on suppose que les activités sont préalablement triées par ordre monotone croissant d'heure de fin, on n'a à traiter chaque activité qu'une seule fois. Il advient souvent qu'un pré traitement de l'entrée ou l'emploi d'une structure de données idoine (une file de priorité, le plus souvent) permet de faire des choix gloutons rapidement et donc d'obtenir un algorithme efficace.

b) Sous-structure optimale

Un problème exhibe une *sous-structure optimale* si une solution optimale du problème contient les solutions optimales des sous-problèmes. Cette propriété est un indice majeur de l'utilisabilité de la programmation dynamique ou des algorithmes gloutons. Comme exemple de sous-structure optimale, rappelons-nous comment nous avons démontré à la section 16.1 que, si une solution optimale du sous-problème S_{ij} incluait une activité a_k , alors elle contenait forcément des solutions optimales pour les sous-problèmes S_{ik} et S_{kj} . Partant de cette sous-structure optimale, nous avons prouvé que, si nous savions quelle activité prendre pour a_k , nous pourrions construire une solution optimale de S_{ij} en sélectionnant a_k plus toutes les activités des solutions optimales des sous-problèmes S_{ik} et S_{kj} . En nous fondant sur cette observation de la sous-structure optimale, nous avons pu concevoir la récurrence (16.3) décrivant la valeur d'une solution optimale.

Nous utilisons généralement une approche plus directe, concernant la sous-structure optimale, quand nous l'appliquons aux algorithmes gloutons. Comme précédemment mentionné, nous pouvons nous permettre le luxe de supposer que nous sommes arrivés à un sous-problème en ayant fait le choix glouton dans le problème originel. Tout ce que nous avons à faire, c'est de prouver qu'une solution optimale du sous-problème, combinée avec le choix glouton déjà effectué, donne une solution optimale du problème original. Cette stratégie utilise implicitement une récurrence sur les sous-problèmes pour prouver que faire le choix glouton à chaque étape produit une solution optimale.

c) Algorithme glouton et programmation dynamique

La propriété de sous-structure optimale étant exploitée à la fois par les stratégies gloutonnes et par la programmation dynamique, on pourrait être tenté de générer une solution par programmation dynamique là où un algorithme glouton suffirait ; on pourrait aussi penser, à tort, qu'une solution gloutonne fonctionne là où la programmation dynamique est nécessaire. Pour illustrer les subtilités entre les deux techniques, intéressons-nous à deux variantes d'un problème classique d'optimisation.

La *variante « entière » du problème du sac-à-dos* est posée de la manière suivante. Un voleur dévalisant un magasin trouve n objets ; le i ème objet vaut v_i euros, et pèse w_i kilogrammes, avec v_i et w_i entiers. Il veut que son butin ait la plus grande valeur possible, mais ne peut pas porter plus de W kilos dans son sac-à-dos, pour un certain entier W . Quels objets devra-t-il prendre ? (Cette variante est dite entière parce que chaque objet doit soit être pris soit abandonné ; le voleur ne peut pas prendre une partie d'objet ni prendre un objet plus d'une fois).

Dans la *variante fractionnaire du problème du sac-à-dos*, le principe est le même, mais le voleur peut prendre des fractions d'objets, au lieu d'avoir un choix binaire (oui ou non) pour chacun. On peut voir un objet de la variante « entière » comme un lingot d'or, et un objet de la variante fractionnaire comme de la poussière d'or.

Les deux problèmes du sac-à-dos exhibent la propriété de sous-structure optimale. Pour le problème entier, on considère le chargement de valeur maximale pesant au plus W kilos. Si l'on retire l'objet j du sac, le chargement restant doit être le meilleur que puisse prendre le voleur pour un poids maximum de $W - w_j$ à partir des $n - 1$ objets initiaux, j étant exclus. Pour le problème fractionnaire, on considère que si l'on retire un poids w d'un objet j dans le chargement optimal, le reste du chargement doit être le meilleur que le voleur puisse emporter pour un poids maximum de $W - w$ à partir des $n - 1$ objets initiaux, et des $w_j - w$ kilos de l'objet j .

Bien que les problèmes soient similaires, la variante fractionnaire peut être résolue par une stratégie gloutonne, contrairement à la variante entière. Pour résoudre le problème fractionnaire, on doit d'abord calculer la valeur par kilo v_i/w_i de chaque objet. En suivant une stratégie gloutonne, le voleur commence par prendre la plus grande quantité possible de l'article ayant la plus grande valeur par kilo. Si cet article ne suffit pas à remplir le sac-à-dos, il prend le plus possible de l'article ayant la plus grande valeur par kilo suivante, et ainsi de suite, jusqu'à ce qu'il ne puisse plus rien emporter. Ainsi, en triant les articles en fonction de leur valeur par kilo, l'algorithme glouton s'exécute en $O(n \lg n)$. La démonstration que la variante fractionnaire du problème du sac-à-dos vérifie bien la propriété du choix glouton est laissée en exercice (Exercice 16.2.1).

Pour comprendre pourquoi cette stratégie gloutonne ne peut pas s'appliquer à la variante entière, considérons l'instance du problème illustrée par la figure 16.2(a). Il existe 3 types d'articles, et le sac peut contenir 50 kilos. L'article 1 pèse 10 kilos et vaut 60 euros. L'article 2 pèse 20 kilos et vaut 100 euros. L'article 3 pèse 30 kilos et vaut 120 euros. Donc, la valeur par kilo de l'article 1 est de 6 euros par kilo, qui est plus grande que celle de l'article 2 (5 euros par kilo) ou de l'article 3 (4 euros par kilo). La stratégie gloutonne ferait donc prendre en premier l'article 1. Néanmoins, comme on peut le voir dans l'analyse de cas de la figure 16.2(b), la solution optimale fait prendre les articles 2 et 3, et délaisser 1. Les deux solutions possibles mettant en jeu l'article 1 ne sont ni l'une ni l'autre optimales.

En revanche, pour la variante fractionnaire du problème, la stratégie gloutonne, qui commence par l'article 1, aboutit à une solution optimale, comme le montre la figure 16.2(c). Prendre l'article 1 ne résout pas la variante entière, parce que le voleur ne peut pas remplir son sac au maximum, et la place vide fait baisser la valeur effective par kilo de son chargement. Dans la variante entière, lorsqu'on envisage de déposer un article dans le sac-à-dos, on doit comparer la solution au sous-problème où l'article est inclus avec celle où l'article est exclu, puis faire un choix. Formulé de cette manière, le problème fait apparaître de nombreux sous-problèmes emboîtés — un indice d'application de la programmation dynamique, et effectivement, la programmation dynamique peut servir à résoudre la variante entière. (Voir l'exercice 16.2.2).

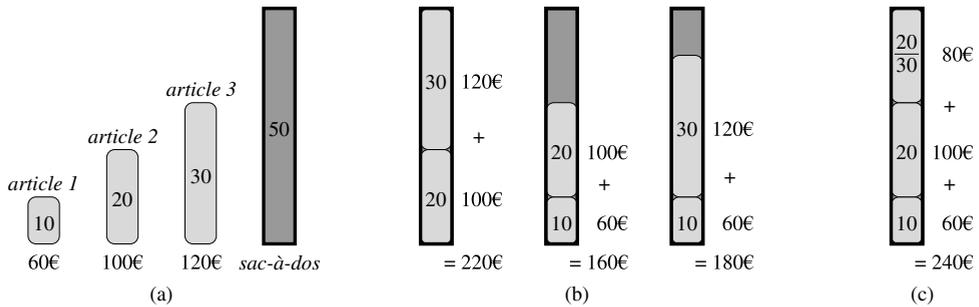


Figure 16.2 La stratégie gloutonne n'est pas adaptée à la variante entière du sac-à-dos. (a) Le voleur doit choisir un sous-ensemble parmi les trois articles montrés, dont le poids ne doit pas excéder 50 kilos. (b) Le sous-ensemble optimal inclut les articles 2 et 3. Une solution quelconque incluant l'article 1 n'est pas optimale, bien que l'article 1 ait la plus grande valeur par kilo. (c) Pour la variante fractionnaire du problème du sac-à-dos, prendre les articles dans l'ordre de la plus grande valeur par kilo aboutit à une solution optimale.

Exercices

16.2.1 Démontrer que la variante fractionnaire du problème du sac-à-dos possède la propriété du choix glouton.

16.2.2 Donner une solution par programmation dynamique à la variante entière du problème du sac-à-dos, qui s'exécute en $O(nW)$, où n est le nombre d'articles différents et W est le poids maximum que le voleur peut mettre dans son sac.

16.2.3 Supposons que dans une variante « entière » du problème du sac-à-dos, l'ordre des articles, quand ils sont triés par poids croissants, soit le même que lorsqu'ils sont triés par valeur décroissante. Donner un algorithme efficace pour trouver une solution optimale à cette variante du problème du sac-à-dos, et montrer pourquoi votre algorithme est correct.

16.2.4 Le Professeur Midas conduit une voiture entre Amsterdam à Lisbonne sur l'Européenne E10. Son réservoir d'essence, quand il est plein, contient assez d'essence pour faire n kilomètres, et sa carte lui donne les distances entre les stations-service sur la route. Le professeur souhaite faire le moins d'arrêts possible pendant le voyage. Donner une méthode efficace grâce à laquelle le Professeur Midas peut déterminer les stations-service où il peut s'arrêter, et démontrer que votre stratégie aboutit à une solution optimale.

16.2.5 Décrire un algorithme efficace qui, étant donné un ensemble $\{x_1, x_2, \dots, x_n\}$ de points sur une droite, détermine le plus petit ensemble d'intervalles fermés de longueur 1 qui contiennent tous les points donnés. Démontrer la validité de votre algorithme.

16.2.6 * Montrer comment résoudre la variante fractionnaire du problème du sac-à-dos en temps $O(n)$.

16.2.7 Soient deux ensembles A et B , contenant chacun n entiers positifs. Vous pouvez choisir de réorganiser chaque ensemble comme vous l'entendez. Après la réorganisation, soit a_i le i ème élément de A et soit b_i le i ème élément de B . Vous recevez alors une indemnité de $\prod_{i=1}^n a_i b_i$. Donner un algorithme qui maximise votre indemnité. Démontrer que votre algorithme maximise l'indemnité et donner son temps d'exécution.

16.3 CODAGES DE HUFFMAN

Les codages de Huffman constituent une technique largement utilisée et très efficace pour la compression de données ; des économies de 20 % à 90 % sont courantes, selon les caractéristiques des données à compresser. Ici, les données sont considérées comme étant une suite de caractères. L'algorithme glouton de Huffman utilise une table contenant les fréquences d'apparition de chaque caractère pour établir une manière optimale de représenter chaque caractère par une chaîne binaire.

Soit un fichier de 100 000 caractères qu'on souhaite conserver de manière compacte. On observe que les caractères du fichier apparaissent avec la fréquence donnée par la figure 16.3. Autrement dit, seulement six caractères différents apparaissent, et le caractère a apparaît 45 000 fois.

	a	b	c	d	e	f
Fréquence (en milliers)	45	13	12	16	9	5
Mot de code de longueur fixe	000	001	010	011	100	101
Mot de code de longueur variable	0	101	100	111	1101	1100

Figure 16.3 Un problème de codage de caractères. Un fichier de données de 100 000 caractères ne contient que les caractères a–f, avec les fréquences indiquées. Si on affecte à chaque caractère un mot de code sur 3 bits, le fichier peut être codé sur 300 000 bits. A l'aide du codage à longueur variable montré ici, le fichier peut être encodé sur 224 000 bits.

On peut représenter ce type de fichier de nombreuses façons. Considérons le problème consistant à déterminer un *codage binaire des caractères* (ou en abrégé *codage*) dans lequel chaque caractère est représenté par une chaîne binaire unique. Si l'on utilise un *codage de longueur fixe*, on a besoin de 3 bits pour représenter six caractères : a = 000, b = 001, ..., f = 101. Cette méthode demande 300 000 bits pour coder entièrement le fichier. Est-il possible de faire mieux ?

Un *codage de longueur variable* peut faire nettement mieux qu'un codage de longueur fixe, en attribuant aux caractères fréquents les mots de code courts et aux caractères moins fréquents les mots de code longs. La figure 16.3 montre ce type de codage ; la chaîne 0 sur 1 bit représente ici a, et la chaîne 1100 sur 4 bits représente f. Ce codage demande

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1\,000 = 224\,000 \text{ bits}$$

pour représenter le fichier, soit une économie d'environ 25 %. En fait, cela représente un codage optimal pour ce fichier, comme nous le verrons.

a) Codages préfixes

Nous ne considérons ici que les codages où aucun mot de code n'est aussi préfixe d'un autre mot du code. Ce type de codages sont dits *préfixes*⁽²⁾. On peut montrer (ce que nous ne ferons pas ici) que la compression de données maximale accessible à l'aide d'un codage de caractères peut toujours être obtenue avec un codage préfixe ; se restreindre aux codages préfixes ne fait donc pas perdre de généralité.

L'encodage est toujours simple pour n'importe quel code de caractères binaire ; on se contente de concaténer les mots de code qui représentent les divers caractères du fichier. Ainsi, avec le code préfixe de longueur variable de la figure 16.3, on code le fichier de 3 caractères abc sous la forme $0 \cdot 101 \cdot 100 = 0101100$, où « \cdot » désigne la concaténation.

Les codages préfixes sont souhaitables car ils simplifient le décodage. Comme aucun mot de code n'est un préfixe d'un autre, le mot de code qui commence un fichier encodé n'est pas ambigu. Il suffit d'identifier le premier mot de code, de le traduire par le caractère initial, de le supprimer du fichier encodé, puis de répéter le processus de décodage sur le reste du fichier. Dans notre exemple, la chaîne 001011101 ne peut être interprétée que comme $0 \cdot 0 \cdot 101 \cdot 1101$, ce qui donne aabe.

Le processus de décodage exige que le codage préfixe ait une représentation commode, de manière qu'on puisse facilement repérer le mot de code initial. Une arborescence binaire dont les feuilles sont les caractères donnés fournit ce genre de représentation. On interprète le mot de code binaire associé à un caractère comme étant le chemin allant de la racine à ce caractère, où 0 signifie « bifurquer vers l'enfant gauche » et 1 signifie « bifurquer vers l'enfant droite ». La figure 16.4 montre les arborescences pour les deux codages de notre exemple. Notez que ce ne sont pas des arborescences binaires de recherche, puisque les feuilles n'ont pas besoin d'être triées et que les nœuds internes ne contiennent pas de clés de caractère.

Un codage optimal pour un fichier est toujours représenté par une arborescence binaire *complète*, dans lequel chaque nœud qui n'est pas une feuille a deux enfants (voir exercice 16.3.1). Le codage de longueur fixe de notre exemple n'est pas optimal puisque son arborescence, montrée à la figure 16.4(a), n'est pas une arborescence binaire complète : certains mots de code commencent par 10... , mais aucun ne commence par 11... . Comme on peut maintenant restreindre notre étude aux arbres binaires complets, on peut dire que, si C est l'alphabet d'où les caractères sont issus et si toutes les fréquences de caractère sont positives, l'arborescence représentant un codage préfixe optimal possède exactement $|C|$ feuilles, une pour chaque lettre de l'alphabet, et exactement $|C| - 1$ nœuds internes (Voir exercice B.5.3).

(2) Sans doute « codages sans préfixes » serait-il plus adapté, mais le terme « codages préfixes » est standard dans la littérature.

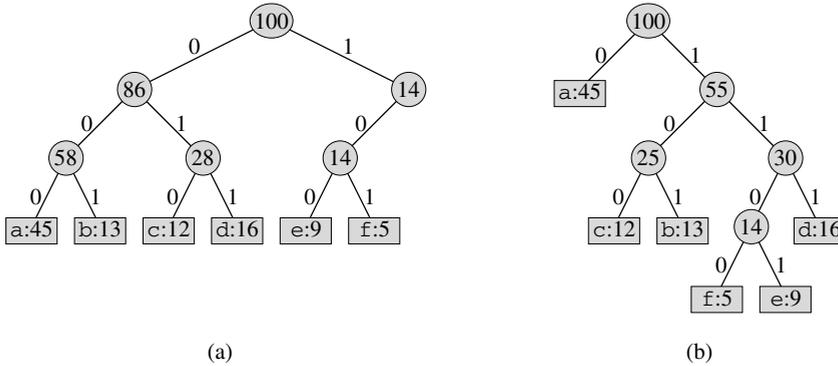


Figure 16.4 Arborescences correspondant aux schémas de codage de la figure 16.3. Chaque feuille est étiquetée avec un caractère et sa fréquence d'apparition. Chaque nœud interne est étiqueté avec la somme des fréquences des feuilles de sa sous-arborescence. (a) L'arborescence correspondant au codage de longueur fixe $a = 000, \dots, f = 101$. (b) L'arborescence correspondant au codage préfixe optimal $a = 0, b = 101, \dots, f = 1100$.

Étant donnée une arborescence T correspondant à un codage préfixe, il est très simple de calculer le nombre de bits nécessaires pour encoder un fichier. Pour chaque caractère c de l'alphabet C , soit $f(c)$ la fréquence de c dans le fichier et soit $d_T(c)$ la profondeur de la feuille c dans l'arbre. Notez que $d_T(c)$ est aussi la longueur du mot de code pour le caractère c .

Le nombre de bits requis pour encoder un fichier vaut donc

$$B(T) = \sum_{c \in C} f(c)d_T(c), \tag{16.5}$$

ce qu'on définit comme étant le **coût** de l'arborescence T .

b) Construction d'un codage de Huffman

Huffman a inventé un algorithme glouton qui construit un codage préfixe optimal appelé **codage de Huffman**.

Nous savons, d'après nos observations de la section 16.2, que la vérification de la validité de l'algorithme repose sur la propriété du choix glouton et sur la sous-structure optimale. Au lieu de démontrer d'abord que ces propriétés sont vérifiées, nous présenterons le pseudo code en premier lieu. Cela nous permettra de clarifier la façon dont l'algorithme fait des choix gloutons.

Dans le pseudo code suivant, on suppose que C est un ensemble de n caractères et que chaque caractère $c \in C$ est un objet possédant une fréquence définie $f[c]$. L'algorithme construit du bas vers le haut l'arborescence T correspondant au codage optimal. Il commence par un ensemble de $|C|$ feuilles et effectue une série de $|C| - 1$ « fusions » pour créer l'arborescence finale. Une file de priorité min F , dont les clés sont prises dans f , permet d'identifier les deux objets les moins fréquents à fusionner. Le résultat de la fusion de deux objets est un nouvel objet dont la fréquence est la somme des fréquences des deux objets fusionnés.

HUFFMAN(C)

```

1   $n \leftarrow |C|$ 
2   $Q \leftarrow C$ 
3  pour  $i \leftarrow 1$  à  $n - 1$ 
4      faire allouer un nouveau nœud  $z$ 
5           $gauche[z] \leftarrow x \leftarrow \text{EXTRAIRE-MIN}(Q)$ 
6           $droite[z] \leftarrow y \leftarrow \text{EXTRAIRE-MIN}(Q)$ 
7           $f[z] \leftarrow f[x] + f[y]$ 
8          INSÉRER( $Q, z$ )
9  retourner EXTRAIRE-MIN( $Q$ )    ▷ Retourner la racine de l'arborescence.

```

Pour notre exemple, l'algorithme de Huffman se déroule comme illustré à la figure 16.5. Comme l'alphabet comprend 6 lettres, la taille initiale de la file est $n = 6$, et 5 étapes de fusion sont nécessaires pour construire l'arborescence. L'arborescence finale représente le codage préfixe optimal. Le mot de code pour une lettre est la séquence d'étiquettes d'arc sur le chemin reliant la racine à la lettre.

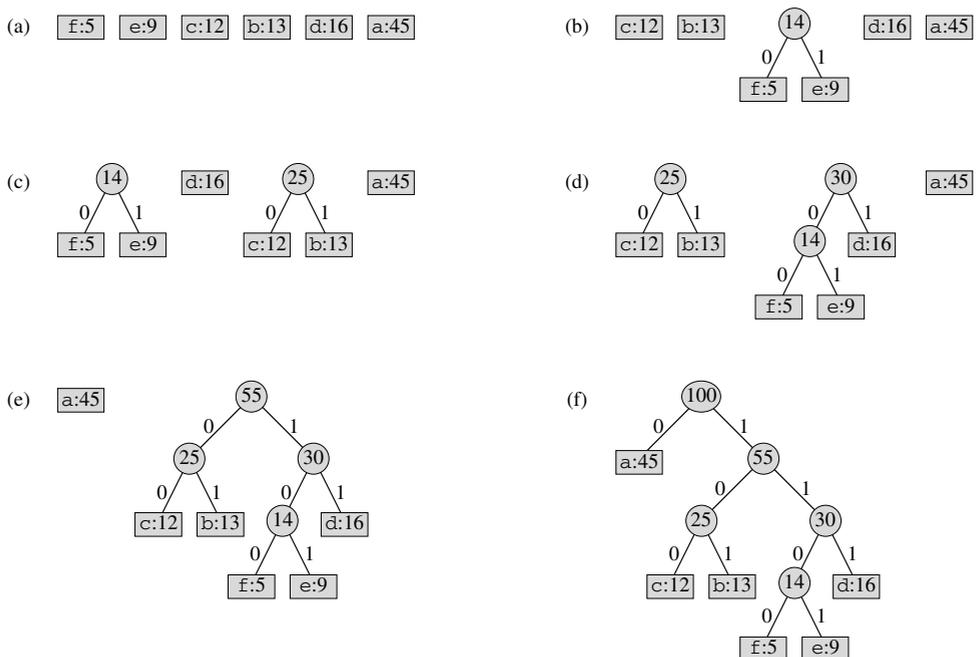


Figure 16.5 Les étapes de l'algorithme de Huffman pour les fréquences données dans la figure 16.3. Chaque partie montre le contenu de la file triée par ordre de fréquence croissante. A chaque étape, les deux arborescences ayant les fréquences les plus basses sont fusionnées. Les feuilles sont représentées par des rectangles contenant un caractère et sa fréquence. Les nœuds internes sont représentés par des cercles contenant la somme des fréquences des enfants. Un arc reliant un nœud interne à ses enfants est étiqueté 0 si c'est un arc vers un enfant gauche, et 1 si c'est un arc vers un enfant droit. Le mot de code pour une lettre est la séquence d'étiquettes des arcs du chemin reliant la racine à la feuille correspondant à cette lettre. (a) L'ensemble initial de $n = 6$ nœuds, un par lettre. (b)–(e) Étapes intermédiaires. (f) L'arborescence finale.

La ligne 2 initialise la file de priorités min F avec les caractères de C . La boucle **pour** des lignes 3–8 extrait plusieurs fois les deux nœuds x et y ayant les fréquences les plus basses de la file, et les remplace dans la file par un nouveau nœud z , résultat de leur fusion. La fréquence de z est calculée comme la somme des fréquences de x et y à la ligne 7. Le nœud z a pour enfant gauche x et pour enfant droit y . (Cet ordre est arbitraire ; échanger les enfants droit et gauche d'un nœud quelconque génère un codage différent de même coût). Après $n - 1$ fusions, le seul nœud qui reste dans la file, à savoir la racine de l'arborescence de codages, est retourné en ligne 9.

L'analyse du temps d'exécution de l'algorithme de Huffman suppose que F est implémentée via un tas min binaire (voir chapitre 6). Pour un ensemble C de n caractères, l'initialisation de F à la ligne 2 peut s'effectuer en temps $O(n)$ via la procédure CONSTRUIRE-TAS de la section 6.3. La boucle **pour** des lignes 3–8 est exécutée exactement $|n| - 1$ fois, et comme chaque opération de tas prend un temps $O(\lg n)$, la boucle contribue pour $O(n \lg n)$ au temps d'exécution. Le temps d'exécution global de HUFFMAN est donc $O(n \lg n)$ sur un ensemble de n caractères.

c) Validité de l'algorithme de Huffman

Pour démontrer que l'algorithme glouton HUFFMAN est correct, on montrera que le problème consistant à déterminer un codage préfixe optimal exhibe les propriétés de choix glouton et de sous-structure optimale. Le lemme suivant montre que la propriété du choix glouton est respectée.

Lemme 16.2 *Soit C un alphabet dans lequel chaque caractère $c \in C$ a une fréquence d'apparition $f[c]$. Soient x et y deux caractères de C ayant les fréquences les plus basses. Il existe alors un codage préfixe optimal pour C dans lequel les mots de code pour x et y ont la même longueur et ne diffèrent que par le dernier bit.*

Démonstration : Le principe de la démonstration est de prendre l'arborescence T représentant un codage préfixe optimal arbitraire et de le modifier pour en faire une arborescence représentant un autre codage préfixe optimal tel que les caractères x et y apparaissent comme des feuilles sœur de profondeur maximale dans la nouvelle arborescence. Si on peut faire cela, alors leurs mots de code auront la même longueur et ne différeront que par le dernier bit.

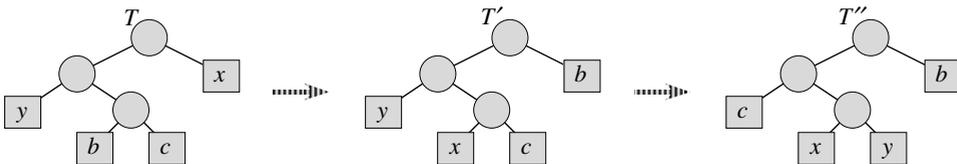


Figure 16.6 Une illustration de l'étape principale de la démonstration du lemme 16.2. Dans l'arborescence optimale T , les feuilles b et c sont deux des feuilles les plus profondes et sont sœurs. Les feuilles x et y sont les deux feuilles fusionnées en premier par l'algorithme de Huffman ; elles apparaissent dans des positions arbitraires de T . Les feuilles b et x sont permutées pour donner l'arborescence T' . Ensuite, les feuilles c et y sont permutées pour obtenir l'arborescence T'' . Comme chaque permutation n'augmente pas le coût, l'arborescence résultant T'' est également une arborescence optimale.

Soient a et b les deux caractères qui sont des feuilles sœurs de profondeur maximale dans T . Sans perdre le caractère général de la démonstration, on suppose que $f[a] \leq f[b]$ et $f[x] \leq f[y]$. Comme $f[x]$ et $f[y]$ sont les deux fréquences de feuille les plus basses, dans l'ordre, et que $f[a]$ et $f[b]$ sont deux fréquences arbitraires, également dans l'ordre, on a $f[x] \leq f[a]$ et $f[y] \leq f[b]$. Comme illustré sur la figure 16.6, on permute les positions dans T de a et x pour produire une arborescence T' , puis on permute les positions dans T' de b et y pour produire une arborescence T'' .

D'après l'équation (16.5), la différence de coût entre T et T' est

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0, \end{aligned}$$

car $f[a] - f[x]$ et $d_T[a] - d_T[x]$ sont tous deux positifs ou nuls. Plus précisément, $f[a] - f[x]$ est positif ou nul parce que x est une feuille de fréquence minimale, et $d_T[a] - d_T[x]$ est positif ou nul parce que a est une feuille de profondeur maximale dans T . De même, comme la permutation de y et b n'augmente pas le coût, $B(T') - B(T'')$ est positif ou nul. Donc, $B(T'') \leq B(T)$, et comme T est optimal, $B(T) \leq B(T'')$, ce qui implique $B(T'') = B(T)$. Donc, T'' est une arborescence optimale dans laquelle x et y sont des feuilles sœur de profondeur maximale, ce qui prouve le lemme. \square

Le lemme 16.2 implique que le déroulement de la construction d'une arborescence optimale par fusions successives peut, sans perte de généralité, commencer par le choix glouton consistant à fusionner les deux caractères ayant les fréquences les plus faibles. Pourquoi est-ce le choix glouton ? On peut voir le coût d'une simple fusion comme la somme des fréquences des deux éléments fusionnés. L'exercice 16.3.3 montrera que le coût total de l'arborescence construite est la somme des coûts de ses fusions. Parmi toutes les fusions possibles à chaque étape, HUFFMAN choisit celle de moindre coût.

Le lemme suivant montre que le problème de la construction d'un codage préfixe optimal vérifie la propriété de sous-structure optimale.

Lemme 16.3 *Soit C un alphabet donné, avec une fréquence $f[c]$ définie pour chaque caractère $c \in C$. Soient x et y deux caractères de C ayant la fréquence minimale. Soit C' l'alphabet C privé des caractères x, y et complété par le (nouveau) caractère z , de sorte que $C' = C - \{x, y\} \cup \{z\}$; définissons f pour C' comme pour C , sauf que $f[z] = f[x] + f[y]$. Soit T' une arborescence représentant un code préfixe optimal pour l'alphabet C' . Alors, l'arborescence T , obtenue à partir de T' en remplaçant le nœud feuille associé à z par un nœud interne ayant x et y comme enfants, représente un code préfixe optimal pour l'alphabet C .*

Démonstration : On commence par montrer que le coût $B(T)$ de l'arborescence T peut être exprimé en fonction du coût $B(T')$ de l'arborescence T' en considérant les coûts qui le composent dans l'équation (16.5). Pour chaque $c \in C - \{x, y\}$, on a $d_T(c) = d_{T'}(c)$, et donc $f[c]d_T(c) = f[c]d_{T'}(c)$. Puisque $d_T(x) = d_T(y) = d_{T'}(z) + 1$, on a

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]), \end{aligned}$$

d'où l'on conclut que

$$B(T) = B(T') + f[x] + f[y].$$

ou, ce qui revient au même,

$$B(T') = B(T) - f[x] - f[y].$$

Nous allons maintenant prouver le lemme en raisonnant par l'absurde. Si T ne représente pas un codage préfixe optimal pour C , il existe une arborescence T'' tel que $B(T'') < B(T)$. Sans nuire à la généralité (d'après le lemme 16.2), T'' a x et y comme frères. Soit T''' l'arborescence T'' dans laquelle le parent commun à x et à y a été remplacé par une feuille z de fréquence $f[z] = f[x] + f[y]$. Alors

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T'), \end{aligned}$$

ce qui contredit l'hypothèse que T' représente un codage préfixe optimal pour C' . Donc, T représente forcément un codage préfixe optimal pour l'alphabet C . \square

Théorème 16.4 *La procédure HUFFMAN produit un codage préfixe optimal.*

Démonstration : Immédiate d'après les lemmes 16.2 et 16.3. \square

Exercices

16.3.1 Démontrer qu'une arborescence binaire qui n'est pas complète ne peut pas correspondre à un codage préfixe optimal.

16.3.2 Donner un codage de Huffman optimal pour l'ensemble de fréquences suivant, basé sur les 8 premiers nombres de Fibonacci ?

$$a : 1 \quad b : 1 \quad c : 2 \quad d : 3 \quad e : 5 \quad f : 8 \quad g : 13 \quad h : 21$$

Pouvez-vous généraliser votre réponse pour trouver le codage optimal lorsque les fréquences sont les n premiers nombres de Fibonacci ?

16.3.3 Démontrer que le coût total d'une arborescence pour un codage particulier peut aussi être calculé comme la somme, prise sur tous les nœuds internes, des fréquences combinées des deux enfants du nœud.

16.3.4 Démontrer que, si les caractères d'un alphabet sont triés par ordre monotone décroissant de fréquences, alors il existe un codage optimal dans lequel les longueurs des mots de code sont monotones croissantes.

16.3.5 On suppose que l'on a un codage préfixe optimal pour un ensemble $C = \{0, 1, \dots, n-1\}$ de caractères, et l'on veut transmettre ce codage en utilisant le minimum de bits. Montrer comment représenter un codage préfixe optimal pour C à l'aide de

$$2n - 1 + n \lceil \lg n \rceil$$

bits seulement. (*Conseil* : Utiliser $2n - 1$ bits pour spécifier la structure de l'arborescence, telle que découverte par un parcours de l'arborescence).

16.3.6 Généraliser l'algorithme de Huffman aux mots de codes ternaires (c'est-à-dire, aux mots de code utilisant les symboles 0, 1 et 2), et démontrer qu'il génère des codages ternaires optimaux.

16.3.7 Soit un fichier de données contenant une séquence de caractères 8 bits, telle que les 256 caractères apparaissent à peu près aussi souvent les uns que les autres : la fréquence de caractère maximale vaut moins de deux fois la fréquence de caractère minimale. Prouver que, dans ce cas, le codage de Huffman n'est pas plus efficace qu'un codage ordinaire de longueur fixe 8 bits.

16.3.8 Montrer qu'aucun schéma de compression ne peut espérer réduire la taille d'un fichier contenant des caractères 8 bits choisis aléatoirement, ne serait-ce que d'un seul bit. (*Conseil* : Comparer le nombre de fichiers avec le nombre de fichiers encodés possibles).

16.4^{*} FONDEMENTS THÉORIQUES DES MÉTHODES GLOUTONNES

Il existe une élégante théorie sur les algorithmes gloutons, que nous allons ébaucher dans cette section. Cette théorie est utile pour déterminer les situations où la méthode gloutonne aboutit à des solutions optimales. Elle s'appuie sur des structures combinatoires connues sous le nom de « matroïdes ». Bien que cette théorie ne couvre pas tous les cas d'application de la méthode gloutonne (par exemple, elle ne prend pas en compte le problème du choix d'activités de la section 16.1 ni le problème du codage de Huffman de la section 16.3), elle couvre de nombreux cas intéressants en pratique. Par ailleurs, cette théorie est en train de se développer et de s'étendre rapidement à nombre d'applications nouvelles ; voir les notes de fin de chapitre pour les références.

16.4.1 Matroïdes

Un *matroïde* est un couple $M = (E, \mathcal{I})$ qui vérifie les conditions suivantes.

- 1) E est un ensemble fini non-vide.
- 2) \mathcal{I} est une famille non vide de sous-ensembles de E , appelée sous-ensembles *indépendants* de E , telle que si $H \in \mathcal{I}$ et $H \subseteq F$, alors $H \in \mathcal{I}$. On dit que \mathcal{I} est *héréditaire* si elle vérifie cette propriété. On remarque que l'ensemble vide \emptyset est obligatoirement membre de \mathcal{I} .
- 3) Si $F \in \mathcal{I}$, $H \in \mathcal{I}$ et $|F| < |H|$, alors il existe un élément $x \in H - F$ tel que $F \cup \{x\} \in \mathcal{I}$. On dit que M vérifie la *propriété d'échange*.

Le mot « matroïde » est dû à Hassler Whitney. Il a étudié les *matroïdes matriciels*, où les éléments de E sont les lignes d'une matrice donnée et où un ensemble de lignes est indépendant si les lignes sont linéairement indépendantes, au sens habituel du terme. On peut facilement montrer que cette structure définit un matroïde (voir exercice 16.4.2).

Comme autre exemple de matroïdes, considérons le *matroïde graphique* $M_G = (E_G, \mathcal{I}_G)$, ainsi défini en fonction d'un graphe non orienté $G = (S, A)$ donné.

- L'ensemble E_G est défini comme étant l'ensemble A des arêtes de G .
- Si F est un sous-ensemble de A , alors $F \in \mathcal{I}_G$ si et seulement si F est acyclique. Autrement dit, un ensemble d'arêtes est indépendant si et seulement si le sous-graphe correspondant forme une forêt.

Le matroïde graphique M_G est très proche du problème de l'arbre couvrant de poids minimale, traité en détail au chapitre 23.

Théorème 16.5 *Si $G = (S, A)$ est un graphe non orienté, alors $M_G = (E_G, \mathcal{I}_G)$ est un matroïde.*

Démonstration : Manifestement, $E_G = A$ est un ensemble fini. De plus, \mathcal{I}_G est héréditaire, puisqu'un sous-ensemble d'une forêt est une forêt. En d'autres termes, supprimer des arêtes dans un ensemble d'arêtes acyclique ne peut pas créer de cycles.

Il reste donc à montrer que M_G vérifie la propriété d'échange. On suppose que $G_F = (S, F)$ et $G_H = (S, H)$ sont des forêts de G et que $|F| > |H|$. Autrement dit, F et H sont des ensembles acycliques d'arête et H contient plus d'arêtes que F .

Il résulte du théorème B.2 qu'une forêt ayant k arêtes contient exactement $|S| - k$ arbres. (Pour le démontrer autrement, on peut partir de $|S|$ arbres, contenant chacun un unique sommet, et d'aucune arête. Dans ce cas, chaque arête ajoutée à la forêt réduit d'une unité le nombre d'arborescences). Donc, la forêt G_F contient $|S| - |F|$ arbres et la forêt G_H contient $|S| - |H|$ arbres.

Comme la forêt G_H possède moins d'arborescences que G_F , elle doit contenir une arborescence T dont les sommets se trouvent dans deux arborescences différentes de la forêt G_F . Par ailleurs, puisque T est connexe, il doit contenir une arête (u, v) telle que

les sommets u et v soient dans des arborescences différentes de la forêt G_F . Comme l'arête (u, v) relie des sommets appartenant à deux arborescences différentes de la forêt G_F , l'arête (u, v) peut être ajoutée à la forêt G_F sans qu'il y ait création de cycle. Donc, M_G vérifie la propriété d'échange, ce qui complète la démonstration établissant que M_G est un matroïde. \square

Étant donné un matroïde $M = (E, \mathcal{I})$, on dit qu'un élément $x \notin F$ est une **extension** de $F \in \mathcal{I}$ si x peut être ajouté à F en préservant l'indépendance ; autrement dit, x est une extension de F si $F \cup \{x\} \in \mathcal{I}$. Considérons l'exemple d'un matroïde graphique M_G . Si F est un ensemble d'arêtes indépendant, alors l'arête e est une extension de F si et seulement si e n'est pas dans F et si l'ajout de e à F ne crée pas de cycle.

Si F est un sous-ensemble indépendant d'un matroïde M , on dit que F est **maximal** s'il ne possède aucune extension. C'est-à-dire s'il n'est contenu dans aucun sous-ensemble indépendant de M plus grand. La propriété suivante est souvent utile.

Théorème 16.6 *Tous les sous-ensembles indépendants maximaux d'un matroïde ont la même taille.*

Démonstration : Supposons au contraire que F soit un sous-ensemble indépendant maximal de M et qu'il en existe un autre H , plus grand. Alors, la propriété d'échange implique que F peut être étendu à un ensemble indépendant $F \cup \{x\}$ pour un certain $x \in H - F$, ce qui contredit l'hypothèse que F est maximal. \square

En guise d'illustration de ce théorème, considérons un matroïde graphique M_G pour un graphe non orienté connexe G . Tout sous-ensemble indépendant maximal de M_G doit être un arbre ayant exactement $|S| - 1$ arêtes, qui relie tous les sommets de G . Une telle arborescence est appelée **arborescence couvrante** de G .

On dit qu'un matroïde $M = (E, \mathcal{I})$ est **pondéré** si l'on dispose d'une fonction de pondération w qui affecte un poids strictement positif $w(x)$ à chaque élément $x \in E$. La fonction de pondération w s'étend aux sous-ensembles de E par sommation :

$$w(F) = \sum_{x \in F} w(x)$$

pour tout $F \subseteq E$. Par exemple, si $w(e)$ désigne la longueur d'une arête e dans un matroïde graphique M_G , alors $w(F)$ est la longueur totale des arêtes de l'ensemble d'arêtes F .

16.4.2 Algorithmes gloutons sur un matroïde pondéré

De nombreux problèmes pour lesquels une approche gloutonne donne des solutions optimales se ramènent à la recherche d'un sous-ensemble indépendant de poids maximal dans un matroïde pondéré. Autrement dit, on dispose d'un matroïde pondéré $M = (E, \mathcal{I})$, et on souhaite trouver un ensemble indépendant $F \in \mathcal{I}$ pour lequel $w(F)$ est maximisé. Un tel sous-ensemble indépendant à pondération maximale est appelé sous-ensemble **optimal** du matroïde. Comme la pondération $w(x)$ d'un élément $x \in E$ quelconque est positive, un sous-ensemble optimal est toujours un sous-ensemble indépendant maximal : il est toujours utile de rendre F le plus grand possible.

Par exemple, dans le *problème de l'arborescence couvrante minimum*, on dispose d'un graphe non-orienté connexe $G = (S, A)$ et d'une fonction longueur w telle que $w(e)$ soit la longueur (positive) de l'arête e . (On utilise le terme « longueur » pour désigner les pondérations initiales des arêtes dans le graphe, en réservant le terme « poids » pour désigner les pondérations dans le matroïde associé). On demande de trouver un sous-ensemble d'arêtes de longueur totale minimale, qui relie tous les sommets. Pour se ramener à un problème de recherche d'un sous-ensemble optimal d'un matroïde, considérons le matroïde pondéré M_G de fonction de pondération w' , où $w'(e) = w_0 - w(e)$ et w_0 est plus grand que la longueur maximale d'une arête. Dans ce matroïde pondéré, tous les poids sont positifs et un sous-ensemble optimal est une arborescence couvrante de longueur totale minimale dans le graphe originel. Plus précisément, chaque sous-ensemble indépendant maximal F correspond à une arborescence couvrante, et comme

$$w'(F) = (|S| - 1)w_0 - w(F)$$

pour tout sous-ensemble indépendant maximal F , un sous-ensemble indépendant qui maximise $w'(F)$ doit minimiser $w(F)$. Donc, tout algorithme capable de trouver un sous-ensemble optimal F dans un matroïde arbitraire peut résoudre le problème de l'arborescence couvrante minimum.

Le chapitre 23 donne des algorithmes adaptés au problème de l'arborescence couvrante minimum, mais ici nous proposons un algorithme glouton qui fonctionne pour un matroïde pondéré quelconque. L'algorithme prend en entrée un matroïde pondéré $M = (E, \mathcal{I})$ associé à une fonction de pondération positive w , et il retourne un sous-ensemble optimal F . Dans notre pseudo code, on désigne les composantes de M par $E[M]$ et $\mathcal{I}[M]$, et la fonction de pondération par w . L'algorithme est glouton parce qu'il considère chaque élément $x \in E$ l'un après l'autre par ordre de poids décroissant et qu'il l'ajoute immédiatement à l'ensemble F en cours de construction si $F \cup \{x\}$ est indépendant.

GLOUTON(M, w)

- 1 $F \leftarrow \emptyset$
- 2 trier $E[M]$ par ordre de poids décroissant w
- 3 **pour** chaque $x \in S[M]$, pris par ordre de poids décroissant $w(x)$
- 4 **faire si** $F \cup \{x\} \in \mathcal{I}[M]$
- 5 **alors** $F \leftarrow F \cup \{x\}$
- 6 **retourner** F

Les éléments de E sont considérés l'un après l'autre, par ordre de poids décroissant. Si l'élément x en cours de traitement peut être ajouté à F sans nuire à l'indépendance de F , il l'est. Sinon, x est écarté. Comme, par définition d'un matroïde, l'ensemble vide est indépendant, et comme x n'est ajouté à F que si $F \cup \{x\}$ est indépendant, le sous-ensemble F est toujours indépendant, par récurrence. Par suite, GLOUTON retourne toujours un sous-ensemble indépendant F . On verra dans un moment que F est un sous-ensemble de poids maximal, et donc que F est un sous-ensemble optimal.

Le temps d'exécution de GLOUTON est facile à analyser. Soit $n = |E|$. La phase de tri de GLOUTON prend un temps $O(n \lg n)$. La ligne 4 est exécutée exactement n fois, une fois pour chaque élément de E . Chaque exécution de la ligne 4 impose de vérifier si l'ensemble $F \cup \{x\}$ est ou non indépendant. Si cette vérification prend un temps $O(f(n))$, l'algorithme tout entier s'exécute en $O(n \lg n + nf(n))$.

Démontrons à présent que GLOUTON retourne un sous-ensemble optimal.

Lemme 16.7 (Les matroïdes vérifient la propriété du choix glouton) *Supposons que $M = (E, \mathcal{I})$ soit un matroïde pondéré associé à la fonction de pondération w , et que E soit trié par ordre de poids décroissant. Soit x le premier élément de E tel que $\{x\}$ soit indépendant, s'il existe un tel x . Si x existe, alors il existe un sous-ensemble optimal F de E qui contient x .*

Démonstration : Si un tel x n'existe pas, alors le seul sous-ensemble indépendant est l'ensemble vide et la démonstration est terminée. Dans le cas contraire, soit H un sous-ensemble non vide optimal. On suppose que $x \notin H$; si tel n'était pas le cas, on ferait $F = H$ et la démonstration s'arrêterait là.

Aucun élément de H n'a un poids supérieur à $w(x)$. Pour le voir, il suffit d'observer que $y \in H$ implique que $\{y\}$ est indépendant, puisque $H \in \mathcal{I}$ et \mathcal{I} est héréditaire. Notre choix de x garantit donc que $w(x) \geq w(y)$ pour tout $y \in H$.

L'ensemble F est construit de la manière suivante. On commence avec $F = \{x\}$. Grâce au choix de x , F est indépendant. En se servant de la propriété d'échange, on trouve itérativement un nouvel élément de H pouvant être ajouté à F jusqu'à ce que $|F| = |H|$, tout en préservant l'indépendance de F . Alors, $F = H - \{y\} \cup \{x\}$ pour un certain $y \in H$, et donc

$$\begin{aligned} w(F) &= w(H) - w(y) + w(x) \\ &\geq w(H). \end{aligned}$$

Comme H est optimal, F l'est forcément aussi ; et comme $x \in F$, le lemme est démontré. \square

On montre ensuite que si un élément n'est pas choisi au départ, il ne le sera jamais.

Lemme 16.8 *Soit $M = (E, \mathcal{I})$ un matroïde. Si x est un élément de E qui est une extension d'un certain sous-ensemble indépendant F de E , alors x est aussi une extension de \emptyset .*

Démonstration : Comme x est extension de F , on sait que $F \cup \{x\}$ est indépendant. Puisque \mathcal{I} est héréditaire, $\{x\}$ doit être indépendant. Par conséquent, x est une extension de \emptyset . \square

Corollaire 16.9 *Soit $M = (E, \mathcal{I})$ un matroïde. Si x est un élément de E tel que x ne soit pas une extension de \emptyset , alors x n'est extension d'aucun sous-ensemble indépendant F de E .*

Démonstration : Ce corollaire est tout simplement la contraposée du lemme 16.8. \square

Le corollaire 16.9 dit qu'un élément qui n'est pas utilisable immédiatement ne le sera jamais. Donc, GLOUTON ne peut pas commettre d'erreur en ignorant les éléments initiaux de E qui ne sont pas des extensions de \emptyset , puisqu'ils ne pourront jamais être utilisés.

Lemme 16.10 (Les matroïdes satisfont la propriété de sous-structure optimale) *Soit x le premier élément de E choisi par GLOUTON pour le matroïde pondéré $M = (E, \mathcal{I})$. Le problème restant, à savoir trouver un sous-ensemble indépendant de poids maximal contenant x , revient à trouver un sous-ensemble indépendant de poids maximal du matroïde pondéré $M' = (E', \mathcal{I}')$, où*

$$\begin{aligned} E' &= \{y \in E : \{x, y\} \in \mathcal{I}\} , \\ \mathcal{I}' &= \{H \subseteq E - \{x\} : H \cup \{x\} \in \mathcal{I}\} , \text{ et} \end{aligned}$$

la fonction de pondération de M' est celle de M , restreinte à E' . (On appelle M' la **contraction** de M par l'élément x).

Démonstration : Si F est un sous-ensemble indépendant de poids maximal de M contenant x , alors $F' = F - \{x\}$ est un sous-ensemble indépendant de M' . Inversement, un sous-ensemble indépendant F' de M' engendre un sous-ensemble indépendant $F = F' \cup \{x\}$ de M . Comme nous avons dans les deux cas $w(F) = w(F') + w(x)$, une solution de poids maximal pour M contenant x donne une solution de poids maximal sur M' , et vice versa. \square

Théorème 16.11 (Validité de l'algorithme glouton sur les matroïdes) *Si $M = (E, \mathcal{I})$ est un matroïde pondéré de fonction de pondération w , alors l'appel GLOUTON(M, w) retourne un sous-ensemble optimal.*

Démonstration : D'après le corollaire 16.9, tous les éléments qui ont été initialement dédaignés parce qu'ils n'étaient pas des extensions de \emptyset peuvent être oubliés pour de bon. Une fois que le premier élément x est sélectionné, le lemme 16.7 implique que GLOUTON ne se trompe pas en ajoutant x à F , puisqu'il existe un sous-ensemble optimal contenant x . Enfin, le lemme 16.10 implique que le reste du problème peut se ramener à trouver un sous-ensemble optimal dans le matroïde M' qui est la contraction de M par x . Après que la procédure GLOUTON a initialisé F à $\{x\}$, toutes les étapes restantes peuvent être interprétées comme agissant sur le matroïde $M' = (E', \mathcal{I}')$, car H est indépendant dans M' si et seulement si $H \cup \{x\}$ est indépendant dans M , pour tous les ensembles $H \in \mathcal{I}'$. Donc, l'action suivante de GLOUTON consistera à trouver un sous-ensemble de M' indépendant de poids maximal, et globalement parlant GLOUTON trouvera un sous-ensemble de M indépendant de poids maximal. \square

Exercices

16.4.1 Montrer que (E, \mathcal{I}_k) est un matroïde, si E est un ensemble fini et \mathcal{I}_k est l'ensemble de tous les sous-ensembles de E de taille au plus égale à k , où $k \leq |E|$.

16.4.2 * Étant donnée une matrice T $m \times n$ sur un certains corps (celui des réels, par exemple), montrer que (E, \mathcal{I}) est un matroïde, où E est l'ensemble des colonnes de T et $F \in \mathcal{I}$, si et seulement si les colonnes de F sont linéairement indépendantes.

16.4.3 * Montrer que si (E, \mathcal{I}) est un matroïde, alors (E, \mathcal{I}') est un matroïde si l'on définit $\mathcal{I}' = \{F' : |F - F'| \text{ contient un certain } F \in \mathcal{I} \text{ maximal}\}$. Autrement dit, les ensembles indépendants maximaux de (E', \mathcal{I}') ne sont autre que les compléments des ensembles indépendants maximaux de (E, \mathcal{I}) .

16.4.4 * Soit E un ensemble fini et E_1, E_2, \dots, E_k une partition de E en sous-ensembles disjoints non vides de E . On définit la structure (E, \mathcal{I}) par la condition $\mathcal{I} = \{F : |F \cap E_i| \leq 1 \text{ pour } i = 1, 2, \dots, k\}$. Montrer que (E, \mathcal{I}) est un matroïde. Autrement dit, l'ensemble de tous les ensembles F qui contiennent au plus un membre de chaque bloc de la partition détermine les ensembles indépendants d'un matroïde.

16.4.5 Montrer comment transformer la fonction de pondération d'un problème de matroïde pondéré, où la solution optimale souhaitée est un sous-ensemble indépendant maximal *de poids minimal*, pour revenir à un problème standard de matroïde pondéré. Argumenter soigneusement la validité de la transformation.

16.5^{*} UN PROBLÈME D'ORDONNANCEMENT DE TÂCHES

Un problème intéressant pouvant être résolu grâce aux matroïdes est le problème de l'ordonnement optimal de tâches de durée unitaire sur un processeur unique, où chaque tâche dispose d'un temps limite de fin d'exécution au delà duquel une pénalité doit être payée. Le problème paraît compliqué, mais on peut le résoudre d'une façon étonnamment simple à l'aide d'un algorithme glouton.

Une *tâche de durée unitaire* est un travail, par exemple un programme à lancer sur un ordinateur, qui demande exactement une unité de temps pour s'exécuter. Étant donné un ensemble fini E de tâches unitaires, un *ordonnement* de E est une permutation de E spécifiant l'ordre dans lequel ces tâches doivent être effectuées. La première tâche de l'ordonnement commence au temps 0 et se termine au temps 1, la deuxième commence au temps 1 et se termine au temps 2, etc.

Voici les entrées du problème de l'*ordonnement de tâches de durée unitaire, avec dates d'échéance et pénalités, sur un processeur unique* :

- un ensemble $E = \{1, 2, \dots, n\}$ de n tâches de durée unitaire ;
- un ensemble de n *dates d'échéance* d_1, d_2, \dots, d_n , tel que chaque d_i vérifie $1 \leq d_i \leq n$ et chaque tâche i est censée se terminer à la date d_i ;
- un ensemble de n *pénalités* positives w_1, w_2, \dots, w_n , tel qu'une pénalité w_i survient si la tâche i n'est pas terminée à la date d_i .

On souhaite trouver un ordonnancement de E qui minimise le total des pénalités.

Considérons un ordonnancement donné. On dit qu'une tâche est **en retard** dans cet ordonnancement si elle se termine après sa date limite. Sinon, la tâche est dite **en avance**. Un ordonnancement arbitraire peut toujours être mis sous la **forme en avance d'abord**, pour laquelle les tâches en avance précèdent les tâches en retard. Pour le voir, il suffit d'observer que, si une tâche en avance a_i suit une tâche en retard a_j , on peut permuter leurs positions sans affecter les caractéristiques de a_i (en avance) et de a_j (en retard).

De même, nous affirmons qu'un ordonnancement arbitraire peut toujours être mis sous **forme canonique**, pour laquelle les tâches en avance précèdent les tâches en retard et les tâches en avance sont ordonnancées par ordre monotone croissant de date d'échéance. Pour ce faire, on donne à l'ordonnancement la forme en-avance-d'abord. Puis, tant qu'il existe deux tâches en avance a_i et a_j se terminant respectivement aux dates k et $k + 1$ dans l'ordonnancement telles que $d_j < d_i$, on permute les positions de a_i et a_j . Comme la tâche a_j est en avance avant la permutation, $k + 1 \leq d_j$. Donc, $k + 1 < d_i$, et donc la tâche a_i est encore en avance après la permutation. La tâche a_j est avancée dans l'ordonnancement, et donc elle reste en avance après la permutation.

La recherche d'un ordonnancement optimal se ramène donc à la recherche d'un ensemble F de tâches qui doivent être en avance dans l'ordonnancement optimal. Une fois F déterminé, on peut créer le véritable ordonnancement en énumérant les éléments de F dans l'ordre croissant de date limite, puis en énumérant les tâches en retard (c'est-à-dire appartenant à $E - F$) dans n'importe quel ordre, ce qui équivaut à un tri canonique de l'ordonnancement optimal.

On dit qu'un ensemble F de tâches est **indépendant** s'il existe un ordonnancement pour ces tâches tel qu'aucune de ces tâches ne soit en retard. Manifestement, l'ensemble des tâches en avance d'un ordonnancement forme un ensemble indépendant de tâches. Soit \mathcal{I} l'ensemble de tous les ensembles indépendants de tâches.

On considère le problème consistant à déterminer si un ensemble de tâches donné F est indépendant. Pour $t = 1, 2, \dots, n$, soit $N_t(F)$ le nombre de tâches de F dont la date d'échéance est inférieure ou égale à t . Notez que $N_0(F) = 0$ pour tout ensemble F .

Lemme 16.12 *Pour tout ensemble de tâches F , les affirmations suivantes sont équivalentes.*

- 1) *L'ensemble F est indépendant.*
- 2) *Pour $t = 1, 2, \dots, n$, on a $N_t(F) \leq t$.*
- 3) *Si les tâches de F sont ordonnancées par ordre croissant de dates d'échéance, alors aucune tâche n'est en retard.*

Démonstration : Visiblement, si $N_t(F) > t$ pour un certain t , alors il est impossible de construire un ordonnancement sans tâches en retard pour l'ensemble F , puisqu'il existe plus de t tâches à effectuer avant le temps t . Donc, (1) implique (2). Si (2) est

satisfaite, alors (3) s'en déduit directement : il est impossible de se tromper en ordonnant les tâches par ordre de dates d'échéance croissantes, puisque (2) implique que la i ème plus grande date d'échéance est au plus égale à i . Enfin, (3) implique (1) de manière triviale. \square

En s'appuyant sur la propriété 2 du lemme 16.12, on peut facilement déterminer si un ensemble de tâches donné est ou non indépendant (voir exercice 16.5.2).

Minimiser la somme des pénalités des tâches en retard revient à maximiser la somme des pénalités des tâches en avance. Le théorème suivant garantit donc que l'algorithme glouton permet de trouver un ensemble indépendant A de tâches dont la somme des pénalités est maximale.

Théorème 16.13 *Si E est un ensemble de tâches de durée unitaire et avec des dates d'échéance, et si \mathcal{I} est l'ensemble de tous les ensembles de tâches indépendants, alors le système (E, \mathcal{I}) correspondant est un matroïde.*

Démonstration : Tout sous-ensemble d'un ensemble de tâches indépendant est obligatoirement indépendant. Pour prouver la propriété d'échange, on suppose que H et F sont des ensembles de tâches indépendants et que $|H| > |F|$. Soit k le plus grand t tel que $N_t(H) \leq N_t(F)$. (Une telle valeur de t existe, vu que $N_0(F) = N_0(H) = 0$.) Puisque $N_n(H) = |H|$ et $N_n(F) = |F|$ mais que $|H| > |F|$, on doit avoir $k < n$ et $N_j(H) > N_j(F)$ pour tout j de l'intervalle $k + 1 \leq j \leq n$. Donc, H contient plus de tâches ayant la date d'échéance $k + 1$ que F . Soit a_i une tâche de $H - F$ ayant la date d'échéance $k + 1$. Soit $F' = F \cup \{a_i\}$.

On va montrer que F' doit être indépendant en utilisant la propriété 2 du lemme 16.12. Pour $0 \leq t \leq k$, on a $N_t(F') = N_t(F) \leq t$, puisque F est indépendant. Pour $k < t \leq n$, on a $N_t(F') \leq N_t(H) \leq t$, puisque H est indépendant. Donc F' est indépendant, ce qui termine la démonstration que (E, \mathcal{I}) est un matroïde. \square

D'après le théorème 16.11, on peut utiliser un algorithme glouton pour trouver un ensemble de tâches F indépendant et de poids maximal. On peut alors créer un ordonnancement optimal en plaçant les tâches de F comme tâches en avance pour cet ordonnancement. Cette méthode est un algorithme efficace pour ordonner des tâches de durée unitaire, avec dates d'échéance et pénalités sur un processeur unique. Le temps d'exécution est $O(n^2)$ avec GLOUTON, puisque chacun des $O(n)$ tests d'indépendance effectués par l'algorithme prend un temps $O(n)$ (voir exercice 16.4). Une implémentation plus rapide est donnée dans le problème 16.5.6.

	Tâche						
	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 16.7 Une instance du problème de l'ordonnement de tâches de durée unitaire, avec dates d'échéance et pénalités sur un processeur unique.

La figure 16.7 donne un exemple d'ordonnement de tâches unitaires avec dates d'échéance et pénalités sur un processeur unique. Dans cet exemple, l'algorithme glouton choisit les tâches a_1 , a_2 , a_3 et a_4 , rejette a_5 et a_6 , puis accepte finalement la tâche a_7 . L'ordonnement optimal final est

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle,$$

pour une pénalité totale de $w_5 + w_6 = 50$.

Exercices

16.5.1 Résoudre l'instance du problème d'ordonnement de la figure 16.7, mais en remplaçant chaque pénalité w_i par $80 - w_i$.

16.5.2 Montrer comment utiliser la propriété 2 du lemme 16.12 pour déterminer en temps $O(|F|)$ si un ensemble donné de tâches F est ou non indépendant.

PROBLÈMES

16.1. Petite monnaie

On considère le problème consistant à rendre n cents en monnaie, en utilisant le moins de pièces possible. On supposera que chaque pièce a une valeur entière.

- Décrire un algorithme glouton permettant de rendre la monnaie en utilisant des pièces de cinquante, vingt, dix, cinq et un cents. Démontrer que votre algorithme aboutit à une solution optimale.
- On suppose que les pièces disponibles pour rendre la monnaie ont des valeurs qui sont des puissances de c , soit p^0, p^1, \dots, p^k , où $p > 1$ et $k \geq 1$ sont deux entiers. Montrer que l'algorithme glouton aboutit toujours à une solution optimale.
- Donner un ensemble de valeurs de pièces pour lequel l'algorithme glouton ne donnera pas de solution optimale. Votre ensemble doit inclure une pièce de 1 cent de façon qu'il y ait une solution pour chaque valeur de n .
- Donner un algorithme à temps $O(nk)$ qui rende la monnaie pour n'importe quel ensemble de k valeurs différentes de pièce, en supposant que l'une des pièces est 1 cent.

16.2. Ordonnement minimisant la durée moyenne d'exécution

On suppose que l'on a un ensemble $S = \{a_1, a_2, \dots, a_n\}$ de tâches, où la tâche a_i exige p_i unités de temps de traitement en tout. On a un ordinateur pour exécuter ces tâches, mais il ne peut exécuter qu'une tâche à la fois. Soit c_i la *date de fin d'exécution* de la tâche a_i , c'est-à-dire l'instant auquel se termine le traitement de la tâche a_i . L'objectif ici est de minimiser la moyenne des dates de fin d'exécution,

c'est-à-dire de minimiser $(1/n) \sum_{i=1}^n c_i$. Par exemple, supposez qu'il y a deux tâches, a_1 et a_2 avec $p_1 = 3$ et $p_2 = 5$, et considérez l'ordonnancement dans lequel a_2 est exécutée en premier, suivie de a_1 . Alors, $c_2 = 5$, $c_1 = 8$, et la durée d'exécution moyenne est $(5 + 8)/2 = 6.5$.

- a. Donner un algorithme qui ordonnance les tâches de façon à minimiser la durée d'exécution moyenne. Chaque tâche doit être exécutée de façon non préemptive : une fois démarrée a_i , elle continue à s'exécuter pendant p_i unités de temps. Prouver que votre algorithme minimise la durée moyenne des dates de fin d'exécution et donner la complexité de votre algorithme.
- b. On suppose maintenant que les tâches ne sont pas toutes disponibles immédiatement : chaque tâche a une **date de disponibilité** r_i avant laquelle elle ne peut pas être traitée. On suppose aussi que l'on autorise la **préemption** : une tâche peut être mise en sommeil, puis repartir ultérieurement. Par exemple, la tâche a_i ayant la durée de traitement $p_i = 6$ peut commencer son exécution à la date 1, être préemptée à la date 4, reprendre son exécution à la date 10, être préemptée de nouveau à la date 11, repartir à la date 13 et enfin s'achever à la date 15. La tâche a_i s'est exécuté en tout pendant 6 unités de temps, mais sa durée d'exécution a été divisée en trois tranches. On dira que la durée d'exécution de a_i est 15. Donner un algorithme qui planifie les tâches de façon à minimiser la durée moyennes des dates de fin d'exécution dans le cadre de ce nouveau scénario. Prouver que votre algorithme minimise la durée moyenne des dates de fin d'exécution et donner son temps d'exécution.

16.3. Sous-graphes acycliques

- a. Soit $G = (S, A)$ un graphe non-orienté. A l'aide de la définition d'un matroïde, montrer que (A, \mathcal{I}) est un matroïde, où $F \in \mathcal{I}$ si et seulement si F est un sous-ensemble acyclique de A .
- b. La **matrice d'incidence** d'un graphe non orienté $G = (S, A)$ est une matrice M de dimension $|S| \times |A|$ telle que $M_{ve} = 1$ si l'arête e est incidente au sommet v , et $M_{ve} = 0$ sinon. Montrer qu'un ensemble de colonnes de M est linéairement indépendant sur le corps des entiers modulo 2 si et seulement si l'ensemble correspondant d'arête est acyclique. Ensuite, se servir du résultat de l'exercice 16.4.2 pour fournir une autre démonstration que le (A, \mathcal{I}) de la partie (a) est un matroïde.
- c. On suppose qu'une pondération positive $w(e)$ est associée à chaque arête d'un graphe non-orienté $G = (S, A)$. Donner un algorithme efficace permettant de trouver un sous-ensemble acyclique de A de poids total maximal.
- d. Soit $G = (S, A)$ un graphe orienté arbitraire, et soit (A, \mathcal{I}) défini de manière que $F \in \mathcal{I}$ si et seulement si F ne contient pas de circuit. Donner un exemple de graphe orienté G tel que le système associé (A, \mathcal{I}) ne soit pas un matroïde. Spécifier quelle est celles des conditions de définition d'un matroïde qui n'est pas satisfaite.
- e. La **matrice d'incidence** d'un graphe orienté $G = (S, A)$ est une matrice M de dimension $|S| \times |A|$ telle que $M_{ve} = -1$ si l'arc e part du sommet v , $M_{ve} = 1$ si l'arc e

arrive au sommet v , et $M_{ve} = 0$ sinon. Montrer que, si un ensemble de colonnes de M est linéairement indépendant, alors l'ensemble d'arcs correspondant ne contient pas de circuit.

- f.* L'exercice 16.4.2 nous dit que l'ensemble des ensembles de colonnes linéairement indépendants d'une matrice M forme un matroïde. Expliquer soigneusement pourquoi les résultats des parties (d) et (e) ne sont pas contradictoires. Comment se fait-il qu'il n'y ait pas de correspondance parfaite entre la notion d'ensemble d'arcs ne formant pas de circuit et la notion d'ensemble linéairement indépendant associé de colonnes de la matrice d'incidences ?

16.4. Variantes du problème de l'ordonnancement

Considérons l'algorithme suivant pour résoudre le problème de la section 16.5 consistant à ordonnancer des tâches unitaires avec dates d'échéance et pénalités. On admet qu'au départ les n intervalles de temps sont tous vides, l'intervalle de temps i étant l'intervalle de longueur unitaire qui se termine au temps i . On considère les tâches par ordre monotone décroissant des pénalités. Quand on traite la tâche a_j , s'il existe des intervalles temporels non encore affectés qui sont situés à ou avant la date d'échéance d_j de a_j , on affecte a_j au dernier de ces intervalles, ce qui a pour effet d'initialiser l'intervalle. S'il n'existe plus d'intervalle de ce type, on affecte la tâche a_j au dernier des intervalles présentement non affectés.

- a.* Dire pourquoi cet algorithme donne toujours une réponse optimale.
- b.* Utiliser la forêt d'ensembles disjoints rapide de la section 21.3 pour implémenter l'algorithme efficacement. On suppose que l'ensemble des tâches en entrée est déjà trié par ordre monotone décroissant de pénalité. Analyser le temps d'exécution de votre implémentation.

NOTES

On trouvera beaucoup plus de données sur les algorithmes gloutons et les matroïdes dans Lawler [196], ainsi que dans Papadimitriou et Steiglitz [237].

L'algorithme glouton apparut d'abord dans la littérature de l'optimisation combinatoire en 1971, dans un article de Edmonds [85], bien que la théorie des matroïdes remonte à un article paru en 1935 et dû à Whitney [314].

Notre démonstration de la validité de l'algorithme glouton pour le problème du choix d'activités s'inspire de celle de Gavril [112]. Le problème de l'ordonnancement de tâches est étudié par Lawler [196], par Horowitz et Sahni [157], ainsi que par Brassard et Bratley [47].

Les codages de Huffman furent inventés en 1952 [162] ; Lelewer et Hirschberg [200] proposent un survol des techniques de compression de données parues avant 1987.

Une extension de la théorie des matroïdes, à savoir la théorie des « gloutonoïdes » fut initiée par Korte et Lovász [189, 190, 191, 192], qui ont largement généralisé la théorie présentée ici.

Chapitre 17

Analyse amortie

Dans une *analyse amortie*, le temps requis pour effectuer une suite d'opérations sur une structure de données est une moyenne sur l'ensemble des opérations effectuées. L'analyse amortie permet de montrer que le coût moyen d'une opération est faible si l'on établit sa moyenne sur une suite d'opérations, même si l'opération prise individuellement est coûteuse. L'analyse amortie diffère de l'analyse du cas moyen au sens où on ne fait pas appel aux probabilités : une analyse amortie garantit les *performances moyennes de chaque opération dans le cas le plus défavorable*.

Les trois premières sections de ce chapitre couvrent les trois techniques les plus communément utilisées en analyse amortie. La section 17.1 commence par la méthode de l'agrégat, dans laquelle on détermine un majorant $T(n)$ sur le coût total d'une suite de n opérations. Le coût moyen par opération est alors $T(n)/n$. On prend comme coût amorti par opération le coût moyen, de façon que toutes les opérations aient le même coût amorti.

La section 17.2 couvre la méthode comptable, dans laquelle on détermine un coût amorti pour chaque opération. Quand il existe plus d'un type d'opération, chacun peut avoir un coût amorti différent. La méthode comptable surfacture certaines opérations au début de la séquence, cette surfacturation servant d'« arrhes » pour des objets particuliers de la structure de données. Le crédit sera utilisé plus tard dans la séquence pour payer des opérations qui sont sous-facturées.

La section 17.3 étudie la méthode du potentiel, qui ressemble à la méthode comptable par le fait qu'on détermine le coût amorti de chaque opération, et qu'on peut

surfacturer certaines opérations au début pour compenser des sous-facturations ultérieures. La méthode du potentiel gère le crédit comme « énergie potentielle » de la structure de données globale, au lieu de l'associer à des objets individuels de la structure de données.

Nous utiliserons deux exemples pour étudier ces trois modèles. Le premier est une pile à laquelle on ajoute l'opération MULTIDÉP, qui dépile plusieurs objets à la fois. Le second est un compteur binaire qui compte à partir de 0 au moyen de la seule opération INCRÉMENTER.

En lisant ce chapitre, gardez à l'esprit que les charges affectées lors d'une analyse amortie n'ont de sens que pour l'analyse elle-même. Elles n'ont pas à apparaître dans le code. Si, par exemple, un crédit est affecté à un objet x par la méthode comptable, il est inutile d'affecter la quantité appropriée à un attribut *crédit*[x] dans le code.

Le point de vue acquis en effectuant une analyse amortie sur une structure de données particulière peut aider à optimiser la conception. À la section 17.4, par exemple, nous utiliserons la méthode du potentiel pour analyser une table capable de s'étendre ou de se contracter dynamiquement.

17.1 MÉTHODE DE L'AGRÉGAT

Dans la *méthode de l'agrégat* on montre que, pour tout n , une suite de n opérations prend le temps total $T(n)$ dans le cas le plus défavorable. Dans le cas le plus défavorable le coût moyen, ou *coût amorti*, par opération est donc $T(n)/n$. Notez que ce coût amorti s'applique à chaque opération, même quand il existe plusieurs types d'opérations dans la séquence. Les deux autres méthodes que nous étudierons pourront affecter des coûts amortis différents aux différents types d'opérations.

a) Opérations de pile

Dans notre premier exemple d'analyse via méthode de l'agrégat, on analyse des piles qui ont été étendues avec une nouvelle opération. La section 10.1 présentait les deux opérations fondamentales de pile qui prenaient chacune un temps $O(1)$:

EMPILER(S, x) empile l'objet x sur la pile S .

DÉPILER(S) dépile le sommet de la pile S et retourne l'objet dépilé.

Comme chacune de ces opérations s'exécute en $O(1)$, considérons qu'elles ont toutes les deux un coût égal à 1. Le coût total d'une suite de n opérations EMPILER et DÉPILER vaut donc n , et le temps d'exécution réel pour n opérations est donc $\Theta(n)$.

Ajoutons l'opération MULTIDÉP(S, k), qui retire les k premiers objets du sommet de la pile S , ou dépile la pile toute entière si elle contient moins de k objets. Dans le pseudo code suivant, l'opération PILE-VIDE retourne VRAI s'il n'y a plus aucun objet sur la pile, et FAUX sinon.

```

MULTIDÉP( $S, k$ )
1  tant que pas PILE-VIDE( $S$ ) et  $k \neq 0$ 
2    faire DÉPILER( $S$ )
3     $k \leftarrow k - 1$ 

```

La figure 17.1 montre un exemple de l'action de MULTIDÉP.

sommets	→	23		
		17		
		6		
		39		
		10	sommets	→
		47		10
		—		47
		(a)		(b)
				(c)

Figure 17.1 L'action de MULTIDÉP sur une pile S , montrée dans sa configuration de départ en (a). Les 4 objets du sommet sont dépilés par MULTIDÉP($S, 4$), dont le résultat apparaît en (b). L'opération suivante est MULTIDÉP($S, 7$), qui vide la pile (partie (c)), puisqu'il reste moins de 7 objets.

Quel est le temps d'exécution de MULTIDÉP(S, k) sur une pile de s objets ? Le temps d'exécution réel est linéaire par rapport au nombre d'opérations DÉPILER effectivement exécutées, et il suffit donc d'analyser MULTIDÉP en fonction des coûts abstraits de 1 attribués à EMPILER et DÉPILER. Le nombre d'itérations de la boucle **tant que** est le nombre $\min(s, k)$ d'objets retirés de la pile. Pour chaque itération de la boucle, on fait un appel à DÉPILER à la ligne 2. Le coût total de MULTIDÉP est donc $\min(s, k)$, et le temps d'exécution réel est une fonction linéaire de ce coût.

Analysons une suite de n opérations EMPILER, DÉPILER, et MULTIDÉP sur une pile initialement vide. Le coût dans le cas le plus défavorable d'une opération MULTIDÉP dans la séquence est $O(n)$, puisque la taille de la pile est au plus égale à n . Le coût dans le cas le plus défavorable d'une opération de pile quelconque est donc $O(n)$, et une suite de n opérations coûte $O(n^2)$, puisqu'on pourrait avoir $O(n)$ opérations MULTIDÉP coûtant chacune $O(n)$. Bien que cette analyse soit correcte, le résultat $O(n^2)$, obtenu en considérant le coût le plus défavorable de chaque opération individuelle, n'est pas assez fin.

Grâce à la méthode d'analyse par agrégat, on peut obtenir un meilleur majorant, qui prend en compte globalement la suite des n opérations. En fait, bien qu'une seule opération MULTIDÉP soit potentiellement coûteuse, une suite de n opérations EMPILER, DÉPILER et MULTIDÉP sur une pile initialement vide peut coûter au plus $O(n)$. Pourquoi ? Chaque objet peut être dépilé au plus une fois pour chaque empilement de ce même objet. Donc, le nombre de fois que DÉPILER peut être appelée sur une pile non vide, y compris les appels effectués à l'intérieur de MULTIDÉP, vaut au plus le nombre d'opérations EMPILER, qui lui-même vaut au plus n . Pour une

valeur quelconque de n , n'importe quelle suite de n opérations EMPILER, DÉPILER et MULTIDÉP prendra donc au total un temps $O(n)$. Le coût moyen d'une opération est $O(n)/n = O(1)$. Dans l'analyse de l'agrégat, chaque opération se voit affecter du même coût amorti, égal au coût moyen. Dans cet exemple, chacune des trois opérations de pile a donc un coût amorti $O(1)$.

Insistons encore sur le fait que, bien que nous ayons simplement montré que le coût moyen, et donc le temps d'exécution, d'une opération de pile était $O(1)$, nous n'avons fait intervenir aucun raisonnement probabiliste. Nous avons en fait établi une borne $O(n)$ dans le cas le plus défavorable pour une suite de n opérations. La division de ce coût total par n a donné le coût moyen, c'est-à-dire le coût amorti par opération.

b) Incréméntation d'un compteur binaire

Un autre exemple de la méthode de l'agrégat est illustré par le problème consistant à implémenter un compteur binaire sur k bits qui compte positivement à partir de 0. On utilise pour représenter ce compteur un tableau $A[0..k-1]$ de bits, où $\text{longueur}[A] = k$. Un nombre binaire x qui est stocké dans le compteur a son bit d'ordre inférieur dans $A[0]$ et son bit d'ordre supérieur dans $A[k-1]$, de manière que $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Au départ, $x = 0$, et donc $A[i] = 0$ pour $i = 0, 1, \dots, k-1$.

Pour ajouter 1 (modulo 2^k) à la valeur du compteur, on utilise la procédure suivante.

```

INCRÉMENTER(A)
1   $i \leftarrow 0$ 
2  tant que  $i < \text{longueur}[A]$  et  $A[i] = 1$ 
3      faire  $A[i] \leftarrow 0$ 
4       $i \leftarrow i + 1$ 
5  si  $i < \text{longueur}[A]$ 
6      alors  $A[i] \leftarrow 1$ 

```

La figure 17.2 montre ce qui arrive à un compteur binaire quand il est incrémenté 16 fois, en commençant à la valeur 0 et en finissant par la valeur 16. Au début de chaque itération de la boucle **tant que** des lignes 2–4, on souhaite ajouter un 1 à la position i . Si $A[i] = 1$, alors l'addition d'un 1 fait basculer à 0 le bit i et génère une retenue de 1, à ajouter à la position $i + 1$ lors de la prochaine itération de la boucle. Sinon, la boucle se termine ; ensuite, si $i < k$, on sait que $A[i] = 0$, et donc l'ajout d'un 1 à la position i , qui fait passer le 0 à 1, est pris en charge par la ligne 6. Le coût de chaque opération INCRÉMENTER est linéaire par rapport au nombre de bits basculés.

Comme avec l'exemple de la pile, une analyse rapide fournit une borne correcte mais pas assez fine. Une exécution individuelle de INCRÉMENTER prend un temps $\Theta(k)$ dans le cas le plus défavorable, celui où le tableau A ne contient que des 1. Donc, une séquence de n opérations INCRÉMENTER sur un compteur initialement nul prend un temps $O(nk)$ dans le cas le plus défavorable.

Compteur	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total coût
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 Le comportement d'un compteur binaire sur 8 bits quand sa valeur passe de 0 à 16 après une suite de 16 opérations INCRÉMENTER. Les bits qui sont basculés pour atteindre la prochaine valeur sont en gris. Le coût d'exécution des basculements de bit est donné à droite. Notez que le coût total ne vaut jamais plus de deux fois le nombre total d'opérations INCRÉMENTER.

On peut affiner notre analyse pour établir un coût de $O(n)$, dans le pire des cas, pour une suite de n opérations INCRÉMENTER en observant que tous les bits ne basculent pas chaque fois que INCRÉMENTER est appelée. Comme le montre la figure 17.2, $A[0]$ bascule à chaque appel de INCRÉMENTER. Le deuxième bit de poids le plus fort, $A[1]$, ne bascule qu'une fois sur deux : une suite de n opérations INCRÉMENTER sur un compteur initialisé à zéro fait basculer $A[1]$ $\lfloor n/2 \rfloor$ fois. De même, le bit $A[2]$ ne bascule qu'une fois sur quatre, c'est-à-dire $\lfloor n/4 \rfloor$ dans une séquence de n opérations INCRÉMENTER. En général, pour $i = 0, 1, \dots, k-1$, le bit $A[i]$ bascule $\lfloor n/2^i \rfloor$ fois dans une séquence de n opérations INCRÉMENTER sur un compteur initialisé à zéro. Pour $i \geq k$, le bit $A[i]$ ne bascule jamais. Le nombre total de basculements dans la séquence est donc

$$\begin{aligned} \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n, \end{aligned}$$

d'après l'équation (A.6). Le temps d'exécution, dans le cas le plus défavorable, d'une suite de n opérations INCRÉMENTER sur un compteur initialisé à zéro est donc $O(n)$. Le coût moyen de chaque opération, et donc le coût amorti par opération, est $O(n)/n = O(1)$.

Exercices

17.1.1 Si l'on ajoutait aux opérations de pile une opération MULTIEMP qui empile k éléments, est-ce que le coût amorti des opérations de pile aurait encore une borne $O(1)$?

17.1.2 Montrer que, si l'on ajoutait une opération DÉCRÉMENTER à l'exemple du compteur sur k bits, n opérations pourraient coûter jusqu'à $\Theta(nk)$ de temps.

17.1.3 Une suite de n opérations est effectuée sur une structure de données. La i ème opération coûte i si i est une puissance exacte de 2, et 1 sinon. Utiliser la méthode de l'agrégat pour déterminer le coût amorti par opération.

17.2 MÉTHODE COMPTABLE

Dans la *méthode comptable* on affecte des coûts différents à des opérations différentes, certaines opérations recevant un coût supérieur ou inférieur à leur coût réel. Le coût attribué à une opération est son *coût amorti*. Lorsque le coût amorti d'une opération excède son coût réel, la différence est affectée à des objets spécifiques de la structure de données, sous la forme de *crédit*. Ce crédit pourra servir plus tard à aider à payer des opérations dont le coût amorti est inférieur au coût réel. On peut donc voir le coût amorti d'une opération comme une combinaison de coût réel et de crédit déposé ou utilisé. Cela est très différent de la méthode de l'agrégat, pour laquelle toutes les opérations ont le même coût amorti.

Il faut choisir soigneusement les coûts amortis des opérations. Si l'on veut montrer, via analyse par la méthode comptable, que, dans le cas le plus défavorable, le coût moyen par opération est réduit, alors le coût amorti total d'une séquence d'opérations doit être un majorant du coût réel total de la séquence. Par ailleurs, comme pour la méthode de l'agrégat, cette relation doit être valable pour n'importe quelle séquence d'opérations. Si l'on note c_i le coût réel de la i ème opération et \hat{c}_i le coût amorti de la i ème opération, il faut que

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i \quad (17.1)$$

pour toutes les séquences de n opérations. Le crédit total stocké dans la structure de données est la différence entre le coût amorti total et le coût réel total, soit $\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$. D'après l'inégalité (17.1), le crédit total associé à la structure de données doit être en permanence non négatif. Si le crédit total pouvait devenir négatif (résultat d'une sous-facturation des premières opérations de la séquence avec promesse de réapprovisionner le compte ultérieurement), alors les coûts amortis totaux générés à ce moment-là seraient en-dessous des coûts réels totaux ; pour la séquence d'opérations effectuées jusqu'alors, le coût amorti total ne serait pas un majorant du coût réel total. Il faut donc prendre garde à ce que le crédit total stocké dans la structure ne devienne jamais négatif.

a) Opérations de pile

Pour illustrer la méthode comptable d'analyse amortie, revenons à l'exemple de la pile. On se souvient que les coûts réels des opérations étaient :

EMPILER 1 ,
 DÉPILER 1 ,
 MULTIEMP $\min(k, s)$,

où k est l'argument fourni à MULTIEMP et s la taille de la pile au moment de l'appel. Affectons à ces opérations les coûts amortis suivants :

EMPILER 2 ,
 DÉPILER 0 ,
 MULTIEMP 0 .

Notez que le coût amorti de MULTIEMP est une constante (0), alors que le coût réel est variable. Ici les trois coûts amortis sont en $O(1)$ bien que, en général, les coûts amortis des opérations concernées puissent différer asymptotiquement.

Nous allons maintenant montrer qu'il est possible de payer une séquence quelconque d'opérations de pile en facturant les coûts amortis. Supposons que chaque unité de coût soit représentée par une pièce de 1 euro. Nous commençons avec une pile vide. Rappelez-vous l'analogie vue à la section 10.1 entre la structure de données pile et une pile d'assiettes dans une cafétéria. Lorsqu'on empile une assiette, on utilise 1 euro pour payer le coût réel de l'empilement et il nous reste 1 euro (sur les 2 euros facturés) qu'on place sur l'assiette. A chaque instant, toute assiette de la pile contient donc un euro de crédit.

L'euro conservé dans l'assiette est un acompte pour le coût de son dépilement. Quand on exécute une opération DÉPILER, on ne facture rien pour l'opération et l'on paye son coût réel grâce au crédit contenu dans l'assiette. Pour dépiler une assiette, on retire de l'assiette l'euro de crédit que l'on utilise pour payer le coût réel de l'opération. Ainsi, en facturant un peu plus pour l'opération EMPILER, on n'a pas besoin de facturer quoi que ce soit pour l'opération DÉPILER.

Par ailleurs, nous n'avons rien non plus à facturer pour les opérations MULTIEMP. Pour dépiler la première assiette, on prend de l'assiette l'euro de crédit qui sert à payer le coût réel d'un dépilement. Pour dépiler la deuxième assiette, on utilise l'euro placé sur cette assiette, et ainsi de suite. Nous avons donc, à tout instant, suffisamment facturé à l'avance pour pouvoir payer les opérations MULTIEMP. Autrement dit, comme chaque assiette de la pile contient 1 euro de crédit et que la pile contient toujours un nombre non négatif d'assiette, nous sommes sûrs que le montant du crédit est toujours non négatif. Ainsi, pour une séquence *quelconque* de n opérations EMPILER, DÉPILER et MULTIEMP, le coût amorti total est un majorant du coût réel total. Comme le coût amorti total est $O(n)$, il en est de même du coût réel total.

b) Incrémentation d'un compteur binaire

Comme autre illustration de la méthode comptable, on analyse l'opération INCRÉMENTER sur un compteur binaire qui commence à zéro. Comme nous l'avons observé précédemment, le temps d'exécution de cette opération est proportionnel au nombre de bits basculés, que nous utiliserons comme coût dans cet exemple. Représentons une pièce de 1 euro pour représenter chaque unité de coût (ici, le basculement d'un bit).

Pour l'analyse amortie, facturons un coût amorti de 2 euros pour mettre un bit à 1. Lorsqu'un bit est mis à 1, on utilise 1 euro (sur les deux 2 euros facturés) pour payer la modification du bit et l'on place l'autre euro sur le bit comme crédit à utiliser ultérieurement quand le bit repassera à 0. À chaque instant, chaque 1 du compteur dispose d'un euro de crédit ; il est donc inutile de facturer quelque chose pour remettre un bit à 0 (il suffira de payer la réinitialisation avec l'euro placé sur le bit).

Il est maintenant possible de déterminer le coût amorti de INCRÉMENTER. Le coût de réinitialisation des bits dans la boucle **tant que** est payé grâce aux euros placés sur les bits à réinitialiser. À la ligne 6 de INCRÉMENTER, un bit au plus est mis à 1 ; le coût amorti d'une opération INCRÉMENTER vaut donc au plus 2 euros. Le nombre de 1 dans le compteur n'est jamais négatif, et le montant du crédit n'est donc jamais négatif non plus. Ainsi, pour n opérations INCRÉMENTER, le coût amorti total est $O(n)$, ce qui borne le coût total réel.

Exercices

17.2.1 Une séquence d'opérations est effectuée sur une pile dont la taille n'excède jamais k . Toutes les k opérations, on fait une copie de sauvegarde de toute la pile. Montrer que le coût de n opérations de pile, copies de sauvegarde comprises, est $O(n)$, en assignant des coûts amortis idoines aux diverses opérations de pile.

17.2.2 Refaire l'exercice 17.1.3 en suivant une méthode d'analyse comptable.

17.2.3 Supposons qu'on veuille non seulement incrémenter un compteur, mais aussi le remettre à zéro (c'est-à-dire, mettre tous les bits à 0). Montrer comment implémenter un compteur sous la forme d'un tableau de bits pour qu'une séquence quelconque de n opérations INCRÉMENTER et RÉINITIALISER prenne un temps $O(n)$ sur un compteur initialement à zéro. (*Conseil* : Gérer un pointeur pointant vers le 1 de poids fort.)

17.3 MÉTHODE DU POTENTIEL

Au lieu de représenter le travail prépayé sous la forme d'un crédit stocké dans des objets spécifiques de la structure de données, la *méthode du potentiel* représente le travail prépayé sous la forme d'« énergie potentielle », ou plus simplement « potentiel », pouvant servir à payer les opérations futures. Le potentiel est associé à la

structure de données dans son ensemble, et non à des objets particuliers de la structure de données.

La méthode du potentiel fonctionne comme suit. On commence avec une structure de données initiale D_0 sur laquelle sont effectuées n opérations. Pour chaque $i = 1, 2, \dots, n$, soit c_i le coût réel de la i ème opération et D_i la structure de données qui résulte de l'application de la i ème opération à la structure de données D_{i-1} . Une **fonction potentiel** Φ fait correspondre à chaque structure de données D_i un nombre réel $\Phi(D_i)$, qui est le **potentiel** associé à la structure de données D_i . Le **coût amorti** \hat{c}_i de la i ème opération par rapport à la fonction potentiel Φ est défini par

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \quad (17.2)$$

Le coût amorti de chaque opération est donc son coût réel, plus l'augmentation de potentiel due à l'opération. D'après l'équation (17.2), le coût amorti total des n opérations est

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0). \end{aligned} \quad (17.3)$$

La seconde égalité se déduit de l'équation (A.9) (page 1024), puisque les termes $\Phi(D_i)$ s'annulent mutuellement.

Si l'on peut définir une fonction potentiel Φ de manière que $\Phi(D_n) \geq \Phi(D_0)$, alors le coût amorti total $\sum_{i=1}^n \hat{c}_i$ est un majorant du coût réel total. En pratique, on ne sait pas toujours combien d'opérations seront effectuées. Donc, en imposant que $\Phi(D_i) \geq \Phi(D_0)$ pour tout i , on garantit, comme avec la méthode comptable, que le paiement est effectué d'avance. Il est souvent commode de donner à $\Phi(D_0)$ la valeur 0, et de montrer ensuite que $\Phi(D_i) \geq 0$ pour tout i . (Voir exercice 17.3.1 pour une manière simple de gérer les cas pour lesquels $\Phi(D_0) \neq 0$.)

Intuitivement, si la différence de potentiel $\Phi(D_i) - \Phi(D_{i-1})$ de la i ème opération est positive, le coût amorti \hat{c}_i représente une surfacturation de la i ème opération et le potentiel de la structure de données croît. Si la différence de potentiel est négative, le coût amorti représente une sous-facturation de la i ème opération et le coût réel de l'opération est payé par la diminution du potentiel.

Les coûts amortis définis par les équations (17.2) et (17.3) dépendent du choix de la fonction potentiel Φ . Des fonctions potentielles différentes peuvent engendrer des coûts amortis différents, bien qu'ils majorent toujours les coûts réels. Il faut souvent faire des compromis en choisissant une fonction potentiel ; le choix de la fonction potentiel optimale dépend des bornes temporelles souhaitées.

a) Opérations de pile

Pour illustrer la méthode du potentiel, revenons une fois de plus aux opérations de pile EMPILER, DÉPILER et MULTIEMP. On définit la fonction potentiel Φ sur une pile comme étant le nombre d'objets de la pile. Pour la pile vide D_0 initiale, on a $\Phi(D_0) = 0$. Comme le nombre d'objet de la pile n'est jamais négatif, la pile D_i qui résulte de la i ème opération a un potentiel non négatif ; d'où

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0) .\end{aligned}$$

Le coût amorti total, par rapport à Φ , de n opérations représente donc un majorant du coût réel.

Calculons maintenant les coûts amortis des diverses opérations de pile. Si la i ème opération sur une pile contenant s objets est une opération EMPILER, la différence de potentiel est

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s + 1) - s \\ &= 1 .\end{aligned}$$

D'après l'équation (17.2), le coût amorti de EMPILER est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

Si la i ème opération sur la pile est MULTIEMP(S, k) et que $k' = \min(k, s)$ objets sont dépilés, le coût réel de l'opération est k' et la différence de potentiel est

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

Donc, le coût amorti de l'opération MULTIEMP est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

De même, le coût amorti d'une opération DÉPILER ordinaire est 0.

Le coût amorti de chacune des trois opérations est $O(1)$, et le coût amorti total d'une séquence de n opérations est donc $O(n)$. Comme nous avons déjà montré que $\Phi(D_i) \geq \Phi(D_0)$, le coût amorti total de n opérations est un majorant du coût réel total. Le coût, dans le cas le plus défavorable, de n opérations est donc $O(n)$.

b) Incrémentation d'un compteur binaire

Pour illustrer autrement la méthode du potentiel, reprenons l'incrémentation d'un compteur binaire. Cette fois, on définit le potentiel du compteur après la i ème opération INCRÉMENTER comme étant égal à b_i , nombre de 1 dans le compteur après la i ème opération.

Calculons le coût amorti d'une opération INCRÉMENTER. Supposons que le i ème appel INCRÉMENTER réinitialise t_i bits. Le coût réel de l'opération est au plus $t_i + 1$ puisque, en plus de réinitialiser t_i bits, il met un bit au plus à 1. Si $b_i = 0$, alors la i ème opération réinitialise tous les k bits, et donc $b_{i-1} = t_i = k$. Si $b_i > 0$, alors $b_i = b_{i-1} - t_i + 1$. Dans l'un ou l'autre cas, $b_i \leq b_{i-1} - t_i + 1$ et la différence de potentiel est

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i.\end{aligned}$$

Le coût amorti vaut donc

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2.\end{aligned}$$

Si le compteur démarre à zéro, on a $\Phi(D_0) = 0$. Comme $\Phi(D_i) \geq 0$ pour tout i , le coût amorti total d'une séquence de n opérations INCRÉMENTER est un majorant du coût réel total ; et le coût, dans le cas le plus défavorable, de n opérations INCRÉMENTER est donc $O(n)$.

La méthode du potentiel fournit un moyen commode d'analyser le compteur même s'il ne commence pas à zéro. On a au départ b_0 bits égaux à 1 et, après n opérations INCRÉMENTER, on a b_n bits égaux à 1, pour $0 \leq b_0, b_n \leq k$. (Rappelez-vous que k est le nombre de bits du compteur.) L'équation (17.3) peut être réécrite sous la forme

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0). \quad (17.4)$$

On a $\hat{c}_i \leq 2$ pour tout $1 \leq i \leq n$. Puisque $\Phi(D_0) = b_0$ et $\Phi(D_n) = b_n$, le coût réel total de n opérations INCRÉMENTER est

$$\begin{aligned}\sum_{i=1}^n c_i &\leq \sum_{i=1}^n 2 - b_n + b_0 \\ &= 2n - b_n + b_0.\end{aligned}$$

Notez en particulier ceci : comme $b_0 \leq k$, tant que $k = O(n)$, le coût réel total est $O(n)$. En d'autres termes, si l'on exécute au moins $n = \Omega(k)$ opérations INCRÉMENTER, le coût réel total est $O(n)$, quelle que soit la valeur initiale du compteur.

Exercices

17.3.1 Soit une fonction potentiel Φ telle que $\Phi(D_i) \geq \Phi(D_0)$ pour tout i , mais $\Phi(D_0) \neq 0$. Montrer qu'il existe une fonction potentiel Φ' telle que $\Phi'(D_0) = 0$, que $\Phi'(D_i) \geq 0$ pour tout $i \geq 1$ et que les coûts amortis calculés avec Φ' soient les mêmes que ceux calculés avec Φ .

17.3.2 Refaire l'exercice 17.1.3 en utilisant la méthode du potentiel.

17.3.3 On considère une structure de données de tas min binaire classique à n éléments, qui supporte les instructions INSÉRER et EXTRAIRE-MIN dans un temps $O(\lg n)$ dans le cas le plus défavorable. Donner une fonction potentiel Φ telle que le coût amorti de INSÉRER soit $O(\lg n)$ et que le coût amorti de EXTRAIRE-MIN soit $O(1)$, et montrer qu'elle fonctionne.

17.3.4 Quel est le coût total d'exécution de n opérations de pile EMPILER, DÉPILER et MULTIEMP, si l'on suppose que la pile démarre avec s_0 objets et finit avec s_n objets ?

17.3.5 On suppose qu'un compteur a pour valeur de départ, non pas 0 mais un nombre contenant b bits égaux à 1. Montrer que le coût de n opérations INCRÉMENTER est $O(n)$ si $n = \Omega(b)$. (On ne supposera pas que b est constant.)

17.3.6 Montrer comment implémenter une file avec deux piles ordinaires (exercice 10.1.6) de telle manière que le coût amorti de chaque opération ENFILER et DÉFILER soit $O(1)$.

17.3.7 Imaginer une structure de données qui permette les deux opérations suivantes pour un ensemble S d'entiers :

INSÉRER(S, x) insère x dans l'ensemble S .

SUPPRIMER-MOITIÉ-SUPÉRIEURE(S) supprime les $\lceil S/2 \rceil$ plus grands éléments de S .

Expliquer comment implémenter cette structure pour que toute séquence de m opérations soit exécutée en temps $O(m)$.

17.4 TABLES DYNAMIQUES

Dans certaines applications, on ne sait pas à l'avance le nombre d'objets qui seront stockés dans une table. Il arrive qu'on se rende compte, après allocation d'espace pour une table, qu'elle n'est pas suffisamment grande. La table doit alors être ré-allouée avec une taille plus grande, et tous les objets stockés dans la table initiale doivent être copiés dans la nouvelle table. De même, si de nombreux objets ont été supprimés dans une table, il peut être judicieux de réallouer une table de taille plus petite. Dans cette section, nous étudierons ce problème d'extension et de contraction dynamique d'une table. Grâce à l'analyse amortie, nous montrerons que le coût amorti de l'insertion et de la suppression est seulement de $O(1)$, même si le coût réel d'une opération est grand au moment quand elle déclenche une extension ou une contraction. En outre, nous verrons comment garantir que l'espace inutilisé d'une table dynamique n'excède jamais une fraction constante de l'espace total.

On suppose que la table dynamique supporte les opérations INSÉRER-TABLE et SUPPRIMER-TABLE. INSÉRER-TABLE insère dans la table un élément qui occupe une *alvéole* unique, c'est-à-dire un espace prévu pour contenir un élément. De même, SUPPRIMER-TABLE retire un élément de la table, libérant ainsi une alvéole. Les détails de la méthode de structuration des données utilisée pour organiser la table sont sans importance ; on pourrait utiliser une pile (section 10.1), un tas (chapitre 6) ou

une table de hachage (chapitre 11). On pourrait également utiliser un tableau ou une collection de tableaux pour implémenter le stockage des objets, ce que nous avons fait à la section 10.3.

On trouvera commode d'utiliser un concept introduit dans notre analyse du hachage (chapitre 11). On définit le *facteur de remplissage* $\alpha(T)$ d'une table T non vide comme étant le nombre d'éléments stockés dans la table, divisé par la taille (nombre d'alvéoles) de la table. On donne à une table vide (table sans aucun élément) une taille de 0, et on convient que son facteur de remplissage vaut 1. Si le facteur de remplissage d'une table dynamique est minoré par une constante, l'espace inutilisé de la table n'est jamais supérieur à une fraction constante de la quantité totale d'espace.

On commencera par analyser le comportement d'une table dynamique dans laquelle seules des insertions sont effectuées. Nous considérerons ensuite le cas plus général où à la fois des insertions et des suppressions sont autorisées.

17.4.1 Extension d'une table

Supposons qu'un espace de stockage soit alloué pour une table sous forme d'un tableau d'alvéoles. Une table est remplie lorsque toutes les alvéoles ont été utilisées ou, si l'on veut, quand son facteur de remplissage vaut 1.⁽¹⁾Dans certains environnements logiciels, si l'on tente d'insérer un élément dans une table pleine, on provoque systématiquement l'arrêt du programme. Nous supposerons toutefois que notre environnement de programmation, comme beaucoup d'environnements modernes, dispose d'un système de gestion de la mémoire capable d'allouer et de libérer des blocs de mémoire à la demande. Ainsi, quand un élément est inséré dans une table pleine, on peut *étendre* cette table en allouant une nouvelle table, avec plus d'alvéoles que la précédente, puis copier les éléments de l'ancienne vers la nouvelle. C'est parce que l'on veut que la table réside en permanence dans un espace de mémoire d'un seul tenant qu'il faut allouer un nouveau tableau pour la table agrandie et dupliquer dans la nouvelle table le contenu de l'ancienne.

Une heuristique fréquente consiste à allouer une nouvelle table contenant deux fois plus d'alvéoles que l'ancienne. Si seules des insertions sont effectuées, le facteur de remplissage de la table est toujours au moins 1/2 et donc la quantité d'espace gaspillée ne dépasse jamais la moitié de l'espace total alloué pour la table.

Dans le pseudo code suivant, on suppose que T est un objet représentant la table. Le champ $table[T]$ contient un pointeur vers le bloc de mémoire représentant la table. Le champ $num[T]$ contient le nombre d'éléments de la table et le champ $taille[T]$ est le nombre total d'alvéoles. Au départ, la table est vide : $num[T] = taille[T] = 0$.

(1) Dans certaines situations, comme pour une table de hachage en adressage ouvert, on peut considérer qu'une table est remplie si son facteur de remplissage est égal à une certaine constante strictement inférieure à 1. (Voir exercice 17.4.1.)

```

INSÉRER-TABLE( $T, x$ )
1  si  $taille[T] = 0$ 
2    alors allouer  $table[T]$  avec 1 alvéole
3       $taille[T] \leftarrow 1$ 
4  si  $num[T] = taille[T]$ 
5    alors allouer  $2 \cdot taille[T]$  alvéoles pour nouvelle-table
6      insérer tous les éléments de  $table[T]$  dans nouvelle-table
7      libérer  $table[T]$ 
8       $table[T] \leftarrow nouvelle-table$ 
9       $taille[T] \leftarrow 2 \cdot taille[T]$ 
10  insérer  $x$  dans  $table[T]$ 
11   $num[T] \leftarrow num[T] + 1$ 

```

Remarquez que nous avons ici deux procédures « d'insertion » : la procédure INSÉRER-TABLE elle-même, plus *l'insertion élémentaire* dans une table en lignes 6 et 10. On peut analyser le temps d'exécution de INSÉRER-TABLE en fonction du nombre d'insertions élémentaires, en affectant à chaque insertion élémentaire un coût de 1. On suppose que le temps d'exécution réel de INSÉRER-TABLE est linéaire par rapport au temps d'insertion d'un élément individuel, de sorte que le surcoût dû à l'allocation d'une table initiale en ligne 2 est constant et que le surcoût induit par l'allocation et la libération de mémoire aux lignes 5 et 7 est dominé par le coût du transfert des éléments en ligne 6. On appelle *extension* l'événement pour lequel la clause **alors** des lignes 5–9 est exécutée.

Analysons une séquence de n opérations INSÉRER-TABLE sur une table initialement vide. Quel est le coût c_i de la i ème opération ? S'il reste de la place dans la table courante (ou si c'est la première opération), alors $c_i = 1$, puisqu'on n'a besoin d'effectuer qu'une seule insertion élémentaire à la ligne 10. Si la table courante est pleine, en revanche, et qu'une extension a lieu, alors $c_i = i$: le coût est 1 pour l'insertion élémentaire de la ligne 10, plus $i - 1$ pour les éléments qui doivent être copiés à partir de l'ancienne table vers la nouvelle en ligne 6. Si n opérations sont effectuées, le coût le plus défavorable d'une opération est $O(n)$, ce qui conduit à un majorant de $O(n^2)$ pour le temps d'exécution total de n opérations.

Cette borne n'est pas très fine, car le coût de l'extension de la table n'est pas souvent atteint au cours des n opérations INSÉRER-TABLE. Plus précisément, la i ème opération ne déclenche une extension que si $i - 1$ est une puissance exacte de 2. Le coût amorti d'une opération est en fait $O(1)$, comme on peut le montrer grâce à la méthode de l'agrégat. Le coût de la i ème opération est

$$c_i = \begin{cases} i & \text{si } i - 1 \text{ est une puissance exacte de } 2, \\ 1 & \text{sinon.} \end{cases}$$

Le coût total de n opérations INSÉRER-TABLE est donc

$$\begin{aligned} \sum_{i=1}^n c_i &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n, \end{aligned}$$

puisque'il existe au plus n opérations de coût 1 et que les coûts des autres opérations forment une série géométrique. Comme le coût total de n opérations INSÉRER-TABLE est $3n$, le coût amorti d'une opération individuelle est 3.

Grâce à la méthode comptable, on peut se faire une idée de la raison pour laquelle le coût amorti d'une opération INSÉRER-TABLE doit être 3. Intuitivement, chaque élément paye pour 3 insertions élémentaires : sa propre insertion dans la table courante, son déplacement quand la table est étendue et le déplacement d'un autre élément qui a été placé dans la table lors de l'extension. Par exemple, supposons que la taille de la table soit m juste après une extension. Alors, le nombre d'éléments de la table est $m/2$ et la table ne contient aucun crédit. On facture trois euros pour chaque insertion. L'insertion élémentaire qui vient juste après coûte 1 euro. Un autre euro est placé comme crédit sur l'élément inséré. Le troisième euro est placé comme crédit sur l'un des $m/2$ éléments déjà présents dans la table. Le remplissage de la table demande $m/2 - 1$ insertions supplémentaires ; et donc, jusqu'à ce que la table contienne m éléments et soit pleine, chaque élément a un euro à verser pour sa réinsertion pendant l'extension.

La méthode du potentiel peut aussi servir à analyser une séquence de n opérations INSÉRER-TABLE, et nous l'utiliserons à la section 17.4.2 pour concevoir une opération SUPPRIMER-TABLE ayant également un coût amorti de $O(1)$. On commence par définir une fonction potentiel Φ qui vaut 0 juste après une extension, mais qui augmente jusqu'à prendre la taille de la table quand la table est pleine de manière que l'extension suivante puisse être payée par ce potentiel. La fonction

$$\Phi(T) = 2 \cdot \text{num}[T] - \text{taille}[T] \quad (17.5)$$

est une possibilité. Immédiatement après une extension, on a $\text{num}[T] = \text{taille}[T]/2$, et donc $\Phi(T) = 0$ comme souhaité. Juste avant une extension, on a $\text{num}[T] = \text{taille}[T]$, et donc $\Phi(T) = \text{num}[T]$ comme souhaité. La valeur initiale du potentiel est 0 et, comme la table est toujours au moins à moitié pleine, $\text{num}[T] \geq \text{taille}[T]/2$, ce qui implique que $\Phi(T)$ est toujours non négative. Donc, la somme des coûts amortis de n opérations INSÉRER-TABLE est un majorant de la somme des coûts réels.

Pour analyser le coût amorti de la i ème opération INSÉRER-TABLE, on appelle num_i le nombre d'éléments présents dans la table après la i ème opération, taille_i la taille totale de la table après la i ème opération et Φ_i le potentiel après la i ème opération. Au départ, on a $\text{num}_0 = 0$, $\text{taille}_0 = 0$ et $\Phi_0 = 0$.

Si la i ème opération INSÉRER-TABLE ne provoque pas d'extension, alors on a $taille_i = taille_{i-1}$ et le coût amorti de l'opération est

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - taille_i) - (2 \cdot num_{i-1} - taille_{i-1}) \\ &= 1 + (2 \cdot num_i - taille_i) - (2(num_i - 1) - taille_i) \\ &= 3.\end{aligned}$$

Si en revanche la i ème opération déclenche une extension, alors on a $taille_i = 2 \cdot taille_{i-1}$ et $taille_{i-1} = num_{i-1} = num_i - 1$, ce qui implique que $taille_i = 2 \cdot (num_i - 1)$. Le coût amorti de l'opération est donc

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - taille_i) - (2 \cdot num_{i-1} - taille_{i-1}) \\ &= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3.\end{aligned}$$

La figure 17.3 montre les valeurs de num_i , $taille_i$ et Φ_i en fonction de i . Remarquez la façon dont l'augmentation de potentiel paie l'extension de la table.

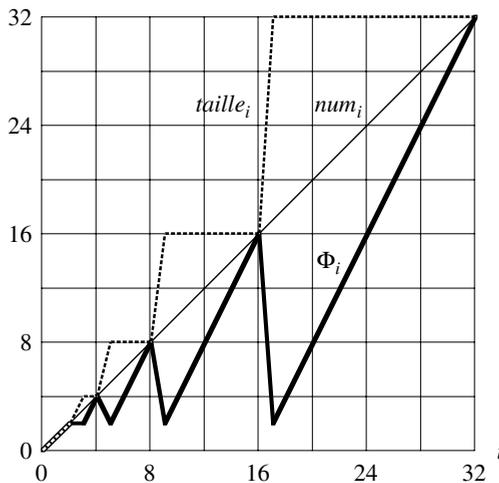


Figure 17.3 L'effet d'une séquence de n opérations INSÉRER-TABLE sur le nombre num_i d'éléments de la table, le nombre $taille_i$ d'alvéoles de la table et le potentiel $\Phi_i = 2 \cdot num_i - taille_i$, chacun étant mesuré après la i ème opération. La courbe fine représente num_i , la courbe en pointillés représente $taille_i$ et la courbe épaisse représente Φ_i . Remarquez que, juste avant une extension, le potentiel a atteint une valeur égale au nombre d'éléments de la table et qu'il peut donc payer le déplacement de tous les éléments vers la nouvelle table. Ensuite, le potentiel retombe à 0 mais augmente immédiatement de 2 lorsque l'élément qui a provoqué l'extension est inséré.

17.4.2 Extension et contraction d'une table

Pour implémenter une opération SUPPRIMER-TABLE, il suffit de retirer de la table l'élément spécifié. Toutefois, il est souvent intéressant de **contracter** la table lorsque le facteur de remplissage devient trop faible, pour que l'espace gaspillé ne soit pas trop important. La contraction d'une table ressemble à son extension : lorsque le nombre d'éléments de la table est trop réduit, on alloue une nouvelle table, plus petite et on copie les éléments de l'ancienne table vers la nouvelle. La libération de l'espace de stockage de l'ancienne table peut alors être confiée au gestionnaire de mémoire. Idéalement, on souhaiterait préserver deux propriétés :

- le facteur de remplissage de la table dynamique est minoré par une constante, et
- le coût amorti d'une opération de table est majoré par une constante.

On suppose que le coût peut être mesuré en termes d'insertions et de suppressions élémentaires.

Une stratégie naturelle pour l'extension et la contraction consiste à doubler la taille de la table lorsqu'un élément est inséré dans une table pleine, et à diminuer de moitié cette taille lorsque le facteur de remplissage de la table devient inférieur à $1/2$ après une suppression. Cette stratégie garantit que le facteur de remplissage n'est jamais inférieur à $1/2$ mais, malheureusement, elle risque de beaucoup augmenter le coût amorti d'une opération. Considérons le scénario suivant. On effectue n opérations sur une table T , où n est une puissance exacte de 2. Les $n/2$ premières opérations sont des insertions, ce qui, d'après notre précédente analyse, coûte au total $\Theta(n)$. A la fin de cette séquence d'insertions, $\text{num}[T] = \text{taille}[T] = n/2$. Pour les $n/2$ opérations restantes, on effectue la séquence suivante :

I, S, S, I, I, S, S, I, I, . . . ,

où I représente une insertion et S une suppression. La première insertion provoque une extension de la table jusqu'à une taille n . Les deux suppressions suivantes provoquent une contraction de la table, qui reprend une taille de $n/2$. Deux insertions supplémentaires causent une nouvelle extension, et ainsi de suite. Le coût de chaque extension ou contraction est $\Theta(n)$, et il y en a $\Theta(n)$. Donc, le coût total des n opérations est $\Theta(n^2)$ et le coût amorti d'une opération est $\Theta(n)$.

La difficulté de cette stratégie est évidente : après une extension, on n'effectue pas assez de suppressions pour payer une contraction. De même, après une contraction, on ne fait pas assez d'insertions pour payer une extension.

Cette stratégie peut être améliorée en permettant au facteur de remplissage de la table de passer sous la barre des $1/2$. Plus précisément, on continue à doubler la taille de la table quand un élément est inséré dans une table pleine ; mais on ne diminue de moitié la taille de la table que quand une suppression provoque un remplissage de la table inférieur à $1/4$ et non à $1/2$ comme précédemment. Le facteur de remplissage de la table est donc minoré par la constante $1/4$. L'idée ici est la suivante : après une extension, le facteur de remplissage vaut $1/2$. Donc, la moitié des éléments de la table

devront être supprimés avant qu'une contraction n'ait lieu, puisque cette contraction ne se produit que si le facteur de remplissage descend en-deçà de $1/4$. De même, après une contraction, le facteur de remplissage vaut encore $1/2$. Donc les insertions éventuelles doivent doubler le nombre d'éléments dans la table avant qu'une extension ait lieu, puisqu'elle ne se produit que lorsque le facteur de remplissage dépasse la valeur 1.

Nous omettrons le code de SUPPRIMER-TABLE, puisqu'il est analogue à INSÉRER-TABLE. Par commodité, on supposera néanmoins dans l'analyse que, si le nombre d'éléments de la table retombe à 0, l'espace de stockage de la table est libéré. Autrement dit, si $num[T] = 0$, alors $taille[T] = 0$.

On peut maintenant utiliser la méthode du potentiel pour analyser le coût d'une séquence de n opérations INSÉRER-TABLE et SUPPRIMER-TABLE. Nous commençons par définir une fonction potentiel Φ qui vaut 0 juste après une extension ou une contraction, et qui augmente quand le facteur de remplissage arrive à 1 ou retombe à $1/4$. Soit $\alpha(T) = num[T]/taille[T]$ le facteur de remplissage d'une table non vide T . Puisque, pour une table vide, $num[T] = taille[T] = 0$ et $\alpha[T] = 1$, on a toujours $num[T] = \alpha(T) \cdot taille[T]$, que la table soit vide ou non. Nous prendrons comme fonction potentiel

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - taille[T] & \text{si } \alpha(T) \geq 1/2, \\ taille[T]/2 - num[T] & \text{si } \alpha(T) < 1/2. \end{cases} \quad (17.6)$$

Observez que le potentiel d'une table vide est 0 et que le potentiel n'est jamais négatif. Donc, le coût amorti total d'une séquence d'opération par rapport à Φ est un majorant du coût réel de la séquence.

Avant de passer à une analyse plus précise, arrêtons-nous pour observer quelques propriétés de cette fonction potentiel. Remarquez que, lorsque le facteur de remplissage vaut $1/2$, le potentiel vaut 0. Quand le facteur de remplissage vaut 1, on a $taille[T] = num[T]$, ce qui implique que $\Phi(T) = num[T]$, et donc que le potentiel peut payer une extension si un élément est inséré. Lorsque le facteur de remplissage vaut $1/4$, on a $taille[T] = 4 \cdot num[T]$, ce qui implique que $\Phi(T) = num[T]$ et donc que le potentiel peut payer le prix d'une contraction si un élément est supprimé. La figure 17.4 illustre le comportement du potentiel pour une séquence d'opérations.

Pour analyser une séquence de n opérations INSÉRER-TABLE et SUPPRIMER-TABLE, soient c_i le coût réel de la i ème opération, \hat{c}_i son coût amorti par rapport à Φ , num_i le nombre d'éléments stockés dans la table après la i ème opération, $taille_i$ la taille totale de la table après la i ème opération, α_i le facteur de remplissage de la table après la i ème opération et Φ_i le potentiel après la i ème opération. Au départ, $num_0 = 0$, $taille_0 = 0$, $\alpha_0 = 1$ et $\Phi_0 = 0$.

Nous commençons par le cas où la i ème opération est INSÉRER-TABLE. L'analyse est la même que celle concernant l'extension de table vue à la section 17.4.1, si $\alpha_{i-1} \geq 1/2$. Que la table soit ou non en phase d'extension, le coût amorti \hat{c}_i de l'opération est au plus 3. Si $\alpha_{i-1} < 1/2$, l'opération ne peut pas provoquer l'extension

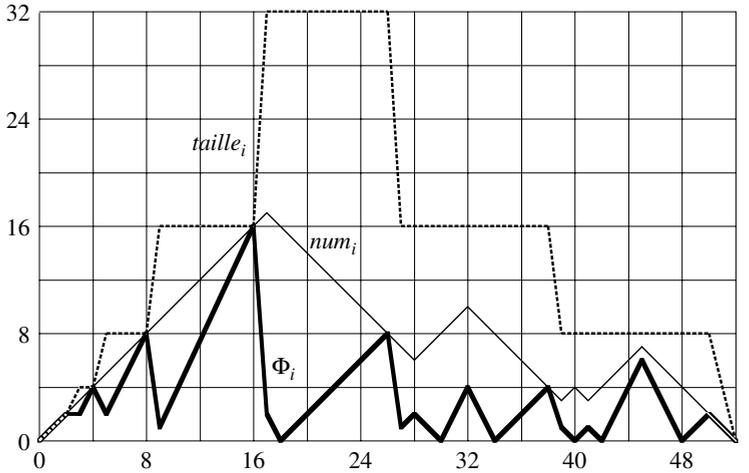


Figure 17.4 L'effet d'une séquence de n opérations INSÉRER-TABLE et SUPPRIMER-TABLE sur le nombre num_i d'éléments de la table, le nombre $taille_i$ d'alvéoles dans la table et le potentiel

$$\Phi_i = \begin{cases} 2 \cdot num_i - taille_i & \text{si } \alpha_i \geq 1/2, \\ taille_i / 2 - num_i & \text{si } \alpha_i < 1/2, \end{cases}$$

chacun étant mesuré après la i ème opération. La courbe fine représente num_i , la courbe en pointillés représente $taille_i$, et la courbe épaisse représente Φ_i . Notez que, juste avant une extension, le potentiel a atteint une valeur égale au nombre d'éléments de la table et qu'il peut donc payer le déplacement de tous les éléments vers la nouvelle table. De même, juste avant une contraction, le potentiel a atteint une valeur égale au nombre d'éléments de la table.

de la table puisque l'extension n'a lieu que lorsque $\alpha_{i-1} = 1$. Si l'on a aussi $\alpha_i < 1/2$, le coût amorti de la i ème opération est

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (taille_i / 2 - num_i) - (taille_{i-1} / 2 - num_{i-1}) \\ &= 1 + (taille_i / 2 - num_i) - (taille_i / 2 - (num_i - 1)) \\ &= 0. \end{aligned}$$

Si $\alpha_{i-1} < 1/2$ mais $\alpha_i \geq 1/2$, alors

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - taille_i) - (taille_{i-1} / 2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - taille_{i-1}) - (taille_{i-1} / 2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} taille_{i-1} + 3 \\ &= 3\alpha_{i-1} taille_{i-1} - \frac{3}{2} taille_{i-1} + 3 \\ &< \frac{3}{2} taille_{i-1} - \frac{3}{2} taille_{i-1} + 3 \\ &= 3. \end{aligned}$$

Le coût amorti d'une opération INSÉRER-TABLE est donc au plus égal à 3.

Penchons-nous maintenant sur le cas où la i ème opération est SUPPRIMER-TABLE. Dans ce cas, $num_i = num_{i-1} - 1$. Si $\alpha_{i-1} < 1/2$, il faut savoir si l'opération provoque ou non une contraction. Si ce n'est pas le cas, $taille_i = taille_{i-1}$ et le coût amorti de l'opération est

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (taille_i / 2 - num_i) - (taille_{i-1} / 2 - num_{i-1}) \\ &= 1 + (taille_i / 2 - num_i) - (taille_i / 2 - (num_i + 1)) \\ &= 2.\end{aligned}$$

Si $\alpha_{i-1} < 1/2$ et que la i ème opération provoque une contraction, le coût réel de l'opération est $c_i = num_i + 1$, car on supprime un élément et on en déplace num_i éléments. On a $taille_i / 2 = taille_{i-1} / 4 = num_{i-1} = num_i + 1$ et le coût amorti de l'opération est

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (taille_i / 2 - num_i) - (taille_{i-1} / 2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1.\end{aligned}$$

Quand la i ème opération est SUPPRIMER-TABLE et que $\alpha_{i-1} \geq 1/2$, le coût amorti est également majoré par une constante. Une analyse est proposée à l'exercice 17.4.2.

En résumé, puisque le coût amorti de chaque opération est majoré par une constante, le temps réel d'une séquence quelconque de n opérations sur une table dynamique est $O(n)$.

Exercices

17.4.1 On souhaite implémenter une table de hachage dynamique à adressage ouvert. Pourquoi pourrait-on vouloir considérer que la table est pleine quand son facteur de remplissage atteint une certaine valeur α strictement inférieure à 1 ? Décrire brièvement la manière de faire des insertions dans une table de hachage dynamique à adressage ouvert, pour qu'elles s'exécutent de manière que la valeur moyenne du coût amorti par insertion soit $O(1)$. Pourquoi la valeur moyenne du coût réel par insertion n'est-elle pas forcément $O(1)$ pour toutes les insertions ?

17.4.2 Montrer que, si la i ème opération sur une table dynamique est SUPPRIMER-TABLE et que $\alpha_{i-1} \geq 1/2$, alors le coût amorti de l'opération par rapport à la fonction potentiel (17.6) est majoré par une constante.

17.4.3 Supposons que, au lieu de contracter une table en diminuant sa taille de moitié quand son facteur de remplissage passe sous la barre des $1/4$, on la contracte en multipliant sa taille par $2/3$ quand son facteur de remplissage tombe en-deçà $1/3$. À l'aide de la fonction potentiel

$$\Phi(T) = |2 \cdot num[T] - taille[T]|,$$

montrer que le coût amorti d'une opération SUPPRIMER-TABLE qui fait appel à cette stratégie est majoré par une constante.

PROBLÈMES

17.1. Compteur binaire de permutations miroir

Le chapitre 30 examine un algorithme important appelé Transformée Rapide de Fourier ou TRF. La première étape de l'algorithme TRF effectue une *permutation miroir* sur un tableau d'entrée $A[0..n-1]$ dont la longueur est $n = 2^k$ pour un certain entier non négatif k . Cette permutation échange les éléments dont les indices ont des représentations binaires qui sont inverses l'une de l'autre.

On peut exprimer chaque indice a comme une séquence de k bits $\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle$, où $a = \sum_{i=0}^{k-1} a_i 2^i$. On définit

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle) = \langle a_0, a_1, \dots, a_{k-1} \rangle ;$$

donc,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

Par exemple, si $n = 16$ (ou, si l'on préfère, $k = 4$), alors $\text{rev}_k(3) = 12$, puisque la représentation sur 4 bits de 3 est 0011, ce qui donne après inversion 1100, représentation sur 4 bits de 12.

- a. Étant donnée une fonction rev_k qui s'exécute dans un temps $\Theta(k)$, écrire un algorithme pour effectuer la permutation miroir sur un tableau de longueur $n = 2^k$ dans un temps $O(nk)$.

On peut utiliser un algorithme basé sur une analyse amortie pour améliorer le temps d'exécution de la permutation miroir. On gère un « compteur de permutations miroir » et une procédure INCRÉMENTER-MIROIR qui, à partir d'une valeur a du compteur de permutations miroir, produit $\text{rev}_k(\text{rev}_k(a) + 1)$. Si $k = 4$, par exemple, et que le compteur commence à 0, alors des appels successifs à INCRÉMENTER-MIROIR produisent la séquence

$$0000, 1000, 0100, 1100, 0010, 1010, \dots = 0, 8, 4, 12, 2, 10, \dots$$

- b. Supposons que les mots de votre ordinateur contiennent des valeurs sur k bits et que, en une unité de temps, l'ordinateur soit capable de manipuler les valeurs binaires avec des opérations comme les décalages droite/gauche d'une quantité arbitraire, le ET logique, le OU logique, etc. Donner une implémentation de la procédure INCRÉMENTER-MIROIR permettant d'effectuer la permutation miroir d'un tableau de n éléments en un temps de $O(n)$ au total.
- c. On suppose qu'il est possible de décaler un mot à gauche ou à droite d'un seul bit en une unité de temps. Est-il encore possible d'implémenter la permutation miroir en $O(n)$?

17.2. Une recherche dichotomique dynamique

La recherche dichotomique dans un tableau trié consomme un temps logarithmique, mais le temps d'insertion d'un nouvel élément est linéaire par rapport à la taille du tableau. On peut améliorer le temps d'insertion en gérant séparément plusieurs tableaux triés.

Plus précisément, on suppose qu'on souhaite implémenter RECHERCHER et INSÉRER sur un ensemble de n éléments. Supposons que $k = \lceil \lg(n+1) \rceil$ et que la représentation binaire de n soit $\langle n_{k-1}, n_{k-2}, \dots, n_0 \rangle$. On a k tableaux triés A_0, A_1, \dots, A_{k-1} où, pour $i = 0, 1, \dots, k-1$, la longueur du tableau A_i est 2^i . Chaque tableau est soit plein, soit vide, selon que $n_i = 1$ ou $n_i = 0$. Le nombre total d'éléments contenus dans les k tableaux est donc $\sum_{i=0}^{k-1} n_i 2^i = n$. Bien que chaque tableau individuel soit trié, il n'existe pas de relation particulière entre les éléments des différents tableaux.

- Décrire comment mettre en œuvre l'opération RECHERCHER sur cette structure de données. Analyser son temps d'exécution dans le cas le plus défavorable.
- Expliquer comment insérer un nouvel élément dans cette structure de données. Analyser son temps d'exécution dans le cas le plus défavorable, puis son temps d'exécution amorti.
- Étudier l'implémentation de SUPPRIMER.

17.3. Arbres équilibrés pondérés amortis

On considère un arbre binaire de recherche ordinaire, étendu par l'ajout à chaque nœud x du champ $taille[x]$ donnant le nombre de clés conservées dans le sous-arbre enraciné en x . Soit α une constante de l'intervalle $1/2 \leq \alpha < 1$. On dit qu'un nœud x donné est α -équilibré si

$$taille[gauche[x]] \leq \alpha \cdot taille[x]$$

et

$$taille[droit[x]] \leq \alpha \cdot taille[x].$$

L'arbre tout entier est dit α -équilibré si tout nœud de l'arbre est α -équilibré. L'approche amortie suivante, qui permet de gérer des arbres équilibrés pondérés, a été suggérée par G. Varghese.

- Un arbre $1/2$ -équilibré est, dans un sens, aussi équilibré que possible. Étant donné un nœud x dans un arbre binaire de recherche quelconque, montrer comment reconstruire le sous-arbre enraciné en x pour qu'il devienne $1/2$ -équilibré. Votre algorithme devra s'exécuter en $\Theta(taille[x])$ et pourra utiliser $O(taille[x])$ espace de stockage auxiliaire.
- Montrer qu'une recherche dans un arbre binaire de recherche α -équilibré à n nœuds prend $O(\lg n)$ dans le cas le plus défavorable.

Pour le reste de ce problème, on suppose que la constante α est strictement supérieure à $1/2$. On admet que INSÉRER et SUPPRIMER sont implémentés de la manière habituelle pour un arbre binaire de recherche de n nœuds hormis que, si après chacune de ces opérations il y a un nœud de l'arbre qui n'est plus α -équilibré, le sous-arbre ayant pour racine le plus haut des nœuds déséquilibrés est « reconstruit » pour devenir $1/2$ -équilibré.

On analysera ce schéma de reconstruction à l'aide de la méthode du potentiel. Pour un nœud x d'un arbre binaire de recherche T , on définit

$$\Delta(x) = |taille[gauche[x]] - taille[droite[x]]| ,$$

et le potentiel de T par

$$\Phi(T) = c \sum_{x \in T: \Delta(x) \geq 2} \Delta(x) ,$$

où c est une constante suffisamment grande qui dépend de α .

- c. Prouver qu'un arbre binaire de recherche quelconque possède un potentiel non négatif et qu'un arbre $1/2$ -équilibré a pour potentiel 0.
- d. On suppose que m unités de potentiel peuvent payer la reconstruction d'un sous-arbre à m nœuds. Quelle doit être la valeur de c en fonction de α pour que la reconstruction d'un sous-arbre qui n'est pas α -équilibré prenne un temps amorti $O(1)$?
- e. Montrer que l'insertion ou la suppression d'un nœud dans un arbre α -équilibré à n nœuds dépense un temps amorti $O(\lg n)$.

17.4. Coût de restructuration d'un arbre rouge-noir

Il existe quatre opérations fondamentales d'arbre rouge-noir qui effectuent des *modifications structurelles* : insertion de nœud, suppression de nœud, rotation et modification de couleur. On a vu que RN-INSÉRER et RN-SUPPRIMER ne consomment que $O(1)$ rotations, insertions de nœud et suppressions de nœud pour préserver les propriétés d'arbre rouge-noir, mais qu'elles risquent de faire beaucoup plus de modifications de couleur.

- a. Décrire un arbre RN valide à n nœuds tel que l'appel de RN-INSÉRER pour ajouter le $(n + 1)$ ème nœud provoque $\Omega(\lg n)$ modifications de couleur. Décrire ensuite un arbre RN valide à n nœuds tel que l'appel de RN-SUPPRIMER sur un nœud particulier provoque $\Omega(\lg n)$ modifications de couleur.

Le nombre de modifications de couleur par opération le plus défavorable qui soit peut être logarithmique, mais nous allons montrer que toute suite de m opérations RN-INSÉRER et RN-SUPPRIMER sur un arbre RN initialement vide provoque $O(m)$ modifications structurelles dans le pire des cas.

- b. Certains des cas traités par la boucle principale des deux procédures RN-INSÉRER-CORRECTION et RN-SUPPRIMER-CORRECTION sont *rédhibitoires* : ils forcent la boucle à se terminer après un nombre constant d'opérations supplémentaires. Pour chacun des cas de RN-INSÉRER-CORRECTION et RN-SUPPRIMER-CORRECTION, spécifier quels sont ceux qui sont rédhibitoires et quels sont ceux qui ne le sont pas. (*Conseil* : Regarder les figures 13.5, 13.6 et 13.7.)

On va analyser d'abord les modifications structurelles quand il n'y a que des insertions. Soient T un arbre RN et $\Phi(T)$ le nombre de nœuds rouges de T . On suppose que 1 unité de potentiel peut payer les modifications structurelles effectuées par l'un quelconque des trois cas de RN-INSÉRER-CORRECTION.

- c. Soit T' le résultat de l'application du cas 1 de RN-INSÉRER-CORRECTION à T . Montrer que $\Phi(T') = \Phi(T) - 1$.
- d. L'insertion d'un nœud dans un arbre RN via RN-INSÉRER peut se diviser en trois parties. Énumérer les modifications structurelles et les changements de potentiel qui résultent des lignes 1–16 de RN-INSÉRER, des cas non rédhibitoires de RN-INSÉRER-CORRECTION et des cas rédhibitoires de RN-INSÉRER-CORRECTION.
- e. En utilisant la partie (d), montrer que le nombre amorti de modifications structurelles faites par un appel RN-INSÉRER est $O(1)$.

On veut maintenant prouver qu'il y a $O(m)$ modifications structurelles quand on fait des insertions et des suppressions. On définit, pour chaque nœud x ,

$$w(x) = \begin{cases} 0 & \text{si } x \text{ est rouge,} \\ 1 & \text{si } x \text{ est noir et n'a pas d'enfant rouge,} \\ 0 & \text{si } x \text{ est noir et a un seul enfant rouge,} \\ 2 & \text{si } x \text{ est noir et a deux enfants rouges.} \end{cases}$$

On redéfinit alors le potentiel d'un arbre RN T par

$$\Phi(T) = \sum_{x \in T} w(x),$$

Soit T' l'arbre qui résulte de l'application d'un des cas non rédhibitoires de RN-INSÉRER-CORRECTION ou RN-SUPPRIMER-CORRECTION à T .

- f. Montrer que $\Phi(T') \leq \Phi(T) - 1$ pour tous les cas non rédhibitoires de RN-INSÉRER-CORRECTION. Prouver que le nombre amorti de modifications structurelles effectuées par un appel RN-INSÉRER-CORRECTION est $O(1)$.
- g. Montrer que $\Phi(T') \leq \Phi(T) - 1$ pour tous les cas non rédhibitoires de RN-SUPPRIMER-CORRECTION. Prouver que le nombre amorti de modifications structurelles effectuées par un appel RN-SUPPRIMER-CORRECTION est $O(1)$.

- h.** Compléter la démonstration que, dans le cas le plus défavorable, une séquence de m opérations RN-INSÉRER et RN-SUPPRIMER fait $O(m)$ modifications structurales.

NOTES

La méthode de l'agrégat pour l'analyse amortie fut utilisée par Aho, Hopcroft et Ullman [5]. Tarjan [293] étudie la méthode comptable et celle du potentiel, et présente plusieurs applications. Il attribue la méthode comptable à plusieurs auteurs, dont M. R. Brown, R. E. Tarjan, S. Huddleston et K. Mehlhorn. Il attribue la méthode du potentiel à D. D. Sleator. Le terme « amorti » est dû à D. D. Sleator et R. E. Tarjan.

Les fonctions potentiel servent aussi à fournir des minorants pour certains types de problèmes. Pour chaque configuration du problème, on définit une fonction potentiel qui associe la configuration à un nombre réel. Ensuite, on détermine le potentiel Φ_{init} de la configuration initiale, le potentiel Φ_{final} de la configuration finale et la variation maximale de potentiel $\Delta\Phi_{\text{max}}$ due à une quelconque étape. Le nombre d'étapes doit donc être au moins $|\Phi_{\text{final}} - \Phi_{\text{init}}| / |\Delta\Phi_{\text{max}}|$. On trouvera des exemples d'utilisation de fonctions potentiel pour le calcul de minorants pour la complexité d'E/S dans les travaux de Cormen [71], Floyd [91] et Aggarwal et Vitter [4]. Krumme, Cybenko et Venkataraman [194] se sont servi de fonctions potentiel pour établir des minorants pour le *comméragage*, c'est-à-dire la diffusion d'un élément unique entre tous les sommets d'un graphe.

PARTIE 5

STRUCTURES DE DONNÉES AVANCÉES

Cette partie traite des structures de données permettant de faire des opérations sur les ensembles dynamiques, mais à un niveau plus avancé que dans la partie 3. Deux des chapitres, par exemple, font un usage intensif des techniques d'analyse amortie vues au chapitre 17.

Le chapitre 18 présente les B-arbres, qui sont des arbres de recherche équilibrés spécialement conçus pour être stockés sur des disques magnétiques. Les disques étant beaucoup plus lents que la mémoire RAM, les performances des B-arbres se mesurent non seulement en fonction du temps de calcul requis par les opérations d'ensembles dynamiques, mais aussi en fonction du nombre d'accès disque effectués. Pour chaque opération de B-arbre, le nombre d'accès disque augmente avec la hauteur du B-arbre, hauteur que les opérations de B-arbre maintiennent faible.

Les chapitres 19 et 20 donnent des implémentations de tas fusionnable qui reconnaissent les opérations INSÉRER, MINIMUM, EXTRAIRE-MIN et UNION. ⁽²⁾ L'opération UNION réunit (fusionne) deux tas. Les structures de données de ces chapitres reconnaissent également les opérations SUPPRIMER et DIMINUER-CLÉ.

Les tas binomiaux, traités au chapitre 19, donnent pour chacune de ces opérations un temps $O(\lg n)$ dans le pire des cas, n étant le nombre total d'éléments du tas donné

(2) Au problème 10.2 on définit un tas fusionnable de façon qu'il reconnaisse MINIMUM et EXTRAIRE-MIN, de sorte que l'on peut parler de *tas min fusionnable*. De même, si le tas reconnaît MAXIMUM et EXTRAIRE-MAX, on parlera de *tas max fusionnable*. Sauf spécification contraire, nos tas fusionnables seront des tas min.

en entrée (des deux tas donnés en entrée pour UNION). Pour l'opération UNION, les tas binomiaux sont meilleurs que les tas binaires du chapitre 6, car la fusion de deux tas binaires nécessite un temps $\Theta(n)$ dans le cas le plus défavorable.

Les tas de Fibonacci, traités au chapitre 20, améliorent les tas binomiaux, du moins en théorie. Leurs performances se mesurent à l'aide de bornes temporelles amorties. Les opérations INSÉRER, MINIMUM et UNION sur les tas de Fibonacci ne prennent que $O(1)$ de temps, réel ou amorti ; les opérations EXTRAIRE-MIN et SUPPRIMER ont pour temps amorti $O(\lg n)$. Toutefois, l'avantage majeur des tas de Fibonacci concerne DIMINUER-CLÉ dont le temps amorti est $O(1)$ seulement. Le faible temps amorti de DIMINUER-CLÉ explique que les tas de Fibonacci soient des composants clé de certains des algorithmes les plus performants asymptotiquement qui sont connus à ce jour en matière de problèmes de graphe.

Enfin, le chapitre 21 présente des structures de données pour les ensembles disjoints. On a un univers de n éléments, que l'on regroupe pour former des ensembles dynamiques. Au départ, chaque élément appartient à son propre singleton. L'opération UNION fusionne deux ensembles et la requête TROUVER-ENSEMBLE identifie l'ensemble qui contient un élément donné à l'instant présent. En représentant chaque ensemble par un arbre enraciné simple, on obtient des opérations étonnamment rapides : une séquence de m opérations s'exécute en un temps $O(m \alpha(n))$, où $\alpha(n)$ est une fonction à croissance très, très, très lente ($\alpha(n)$ vaut au plus 4 dans toutes les applications imaginables). L'analyse amortie qui prouve cette borne temporelle est aussi complexe que la structure de données est simple.

Les thèmes abordés dans cette partie ne sont pas les seuls exemples de structures de données « avancées ». Voici quelques autres exemples de structures de données avancées.

- Les **arbres dynamiques**, introduits par Sleator et Tarjan [281] et étudiés par Tarjan [292], gèrent des forêts d'arbres disjoints. Chaque arc de chaque arbre possède un coût à valeur réelle. Les arbres dynamiques reconnaissent les requêtes permettant de trouver les parents, les racines, les coûts d'arc et le coût d'arc minimal sur un chemin reliant un nœud à la racine. Les manipulations d'arbre vont de la coupure de branche à la transformation d'un nœud en racine, en passant par l'actualisation de tous les coûts d'arc sur un chemin reliant un nœud vers la racine et la connexion d'une racine à un autre arbre. Une implémentation des arbres dynamiques donne une borne temporelle amortie $O(\lg n)$ pour chaque opération ; une autre implémentation, plus compliquée, donne des bornes $O(\lg n)$ dans le pire des cas. Les arbres dynamiques servent aussi dans certains des algorithmes de flot les plus rapides asymptotiquement.
- Les **arbres déployés**, développés par Sleator et Tarjan [282] et étudiés par Tarjan [292], sont une forme d'arbre binaire de recherche sur laquelle les opérations standard d'arbre de recherche s'exécutent avec un temps amorti $O(\lg n)$. L'une des applications des arbres déployés simplifie les arbres dynamiques.

- Les structures de données *persistantes* autorisent les recherches, et parfois aussi les mises à jour, sur d'anciennes versions d'une structure de données. Driscoll, Sarnak, Sleator et Tarjan [82] présentent des techniques permettant de rendre persistantes des structures de données chaînées, avec un coût réduit en temps et en espace. Le problème 13.1. donne un exemple simple d'ensemble dynamique persistant.
- Il existe plusieurs structures de données qui donnent une implémentation plus rapide des opérations de dictionnaire (INSÉRER, SUPPRIMER et RECHERCHER) sur un univers de clés restreint. En exploitant ces restrictions, ces structures donnent de meilleurs temps d'exécution asymptotiques, dans le cas le plus défavorable, que les structures de données basées sur les comparaisons. Une structure inventée par van Emde Boas [301] reconnaît les opérations MINIMUM, MAXIMUM, INSÉRER, SUPPRIMER, RECHERCHER, EXTRAIRE-MIN, EXTRAIRE-MAX, PRÉDÉCESSEUR et SUCCESSEUR avec un temps $O(\lg \lg n)$ dans le cas le plus défavorable, à la condition que l'univers des clés soit restreint à $\{1, 2, \dots, n\}$. Fredman et Willard ont introduit les *arbres de fusion* [99], qui ont été la première structure de données à accélérer les opérations de dictionnaire quand l'univers se limite à des nombres entiers. Ils ont montré comment implémenter ces opérations en un temps $O(\lg n / \lg \lg n)$. Plusieurs structures de données ont suivi, dont les *arbres de recherche exponentiels* [16], qui améliorent encore les bornes pour tout ou partie des opérations de dictionnaire ; ces structures sont mentionnées dans les notes de fin de chapitre tout au long de cet ouvrage.
- Les *structures de données de graphe dynamique* reconnaissent diverses requêtes tout en permettant à la structure d'un graphe d'être modifiée par des opérations qui insèrent ou suppriment des sommets ou des arcs. Exemples de requêtes reconnues : sommet-connexité [144] ; arc-connexité ; arbres couvrant minimaux [143] ; biconnexité ; fermeture transitive [142].

Tout au long de ce livre, les notes de fin de chapitre mentionnent des structures de données complémentaires.

Chapitre 18

B-arbres

Les B-arbres sont des arbres de recherche équilibrés conçus pour être efficaces sur des disques magnétiques ou autres unités de stockage secondaires à accès direct. Les B-arbres ressemblent aux arbres rouge-noir (chapitre 13), mais ils sont plus performants quand il s'agit de minimiser les entrées-sorties disque.

La différence majeure entre les B-arbres et les arbres rouge-noir réside dans le fait que les nœuds des B-arbres peuvent avoir de nombreux enfants, jusqu'à plusieurs milliers. Autrement dit, le « facteur de ramification » d'un B-arbre peut être très grand, bien qu'il soit généralement déterminé par les caractéristiques de l'unité de disque utilisée. Les B-arbres ressemblent aux arbres rouge-noir au sens où tout B-arbre à n nœuds a une hauteur $O(\lg n)$, bien que la hauteur d'un B-arbre puisse être très inférieure à celle d'un arbre rouge-noir car son facteur de ramification peut être beaucoup plus grand. Cela permet donc également d'utiliser les B-arbres pour implémenter en temps $O(\lg n)$ de nombreuses opérations d'ensemble dynamique.

Les B-arbres généralisent de façon naturelle les arbres binaires de recherche. La figure 18.1 montre un B-arbre simple. Si un nœud interne x d'un B-arbre contient $n[x]$ clés, alors x possède $n[x] + 1$ enfants. Les clés du nœud x sont utilisées comme points de séparation de l'intervalle des clés gérées par x en $n[x] + 1$ sous-intervalles, chacun étant pris en charge par un enfant de x . Lorsqu'on recherche une clé dans un B-arbre, on prend une décision à $(n[x] + 1)$ alternatives, via des comparaisons avec les $n[x]$ clés stockées dans le nœud x . La structure des feuilles diffère de celle des nœuds internes ; nous étudierons ces différences à la section 18.1.

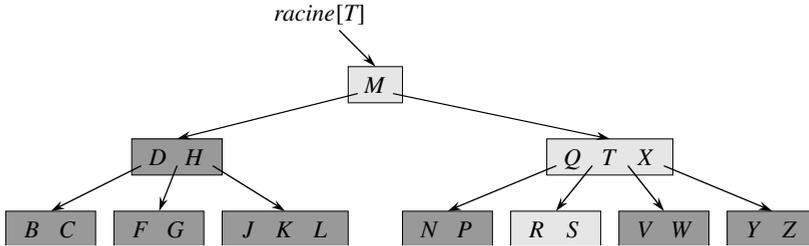


Figure 18.1 Un B-arbre dont les clés forment l'ensemble des consonnes. Un nœud interne x contenant $n[x]$ clés possède $n[x] + 1$ enfants. Toutes les feuilles ont la même profondeur dans l'arbre. Les nœuds parcourus pendant la recherche de la lettre R sont en gris clair.

La section 18.1 donne une définition précise des B-arbres et démontre que la hauteur d'un B-arbre croît seulement de manière logarithmique avec le nombre de nœuds qu'il contient. La section 18.2 décrit comment rechercher ou insérer une clé dans un B-arbre, et la section 18.3 traite de la suppression. Mais avant de continuer, il convient de se demander pourquoi les structures de données adaptées aux disques magnétiques demandent une évaluation différente de celles qui sont conçues pour fonctionner en mémoire principale.

a) Structures de données sur supports de stockage secondaires

Il existe de nombreuses techniques pour fournir de la mémoire à un système informatique. La *mémoire centrale* (ou *mémoire principale*) d'un ordinateur est constituée habituellement de puces de silicium. Le coût de cette technologie (en termes de coût du bit stocké) est de l'ordre de cent fois le coût des stockages magnétiques comme les bandes ou les disques. Un système possède en général de la *mémoire secondaire* sous forme de disques magnétiques ; le volume de stockage de ces supports secondaires est souvent cent à mille fois plus important que la capacité de la mémoire principale.

La figure 18.2 montre un lecteur de disque classique. Le disque se compose de plusieurs *plateaux*, qui tournent à vitesse constante autour d'un axe central commun. La surface de chaque plateau est recouverte de particules magnétisables. Chaque plateau est lu ou écrit par une *tête* située à l'extrémité d'un *bras*. Les bras, qui sont reliés entre eux physiquement, peuvent rapprocher ou éloigner leurs têtes de l'axe central. Quand une tête est stationnaire, la surface qui défile au-dessous d'elle est une *piste*. Les têtes de lecture-écriture sont alignées verticalement en permanence, et donc les pistes qui défilent sous elles sont accédées simultanément. La figure 18.2(b) montre un ensemble de pistes, connu sous le nom de *cylindre*.

Les disques sont meilleur marché et ont plus de capacité que la mémoire centrale, mais ils sont beaucoup, beaucoup plus lents car ils renferment des composants mécaniques mobiles. Il y a deux types de mouvements mécaniques dans un disque : la rotation des plateaux et le déplacement des bras. À l'heure où nous écrivons ces lignes, les disques usuels tournent à des vitesses de 5 400 à 15 000 tours par minute,

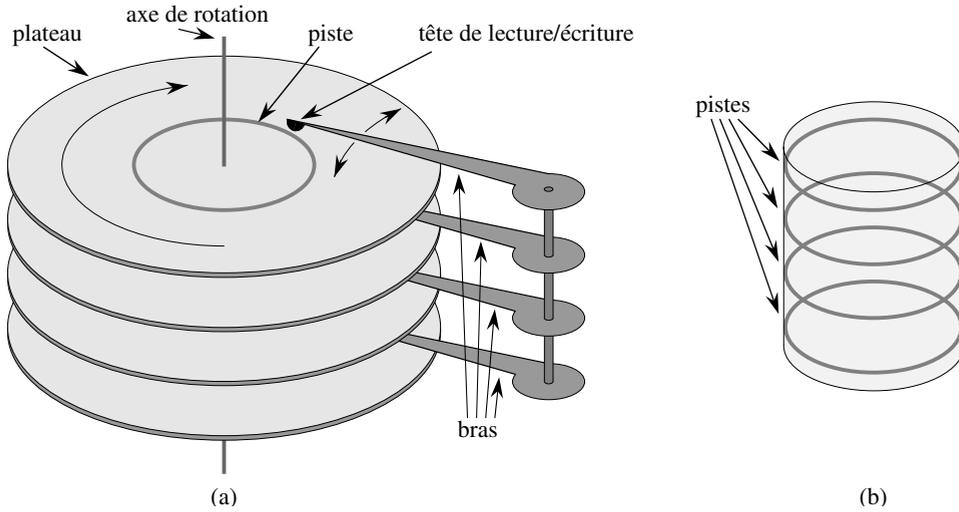


Figure 18.2 (a) Lecteur de disque typique. Il se compose de plusieurs plateaux tournant autour d'un axe central. La lecture et l'écriture de chaque plateau sont faites par une tête située à l'extrémité d'un bras. Les bras sont construits de façon telle que leurs têtes se déplacent à l'unisson. Ici, les bras tournent autour d'un axe pivot commun. Une piste est la surface qui défile sous la tête de lecture-écriture quand celle-ci est stationnaire. (b) Un cylindre se compose d'un ensemble de pistes coaxiales.

7 200 tr/mn étant la valeur la plus courante. 7 200 tr/mn peut sembler rapide ; mais un tour prend 8,33 ms, ce qui est presque cent mille fois plus lent que les temps d'accès de 100 nanosecondes qui est très courant pour les puces de silicium. En d'autres termes, si l'on doit attendre un tour complet pour qu'un certain élément vienne sous la tête de lecture-écriture, pendant ce délai on pourra accéder à la mémoire centrale 100 000 fois ! En moyenne on n'attend que le temps d'une demie révolution, mais la différence de temps d'accès entre puce de silicium et disque reste néanmoins énorme. Le déplacement des bras demande aussi du temps. À l'heure où nous écrivons ces lignes, le temps d'accès moyen pour un disque usuel est de l'ordre de 3 à 9 ms.

Pour amortir le temps passé à attendre les déplacements mécaniques, un disque accède à plusieurs éléments simultanément. Les données sont réparties en un certain nombre de *pages* de bits ayant la même taille et rangées de manière contiguë dans les cylindres ; chaque lecture ou écriture sur le disque implique une ou plusieurs pages toutes entières. Pour un disque typique, une page fait entre 2^{11} et 2^{14} octets de longueur. Une fois que la tête de lecture-écriture est positionnée correctement et que le disque a tourné pour que le début de la page désirée soit sous la tête, la lecture ou l'écriture est entièrement électronique (mis à part la rotation du disque), et il est alors possible de transférer rapidement de gros volumes de données.

Souvent, il est plus long pour l'ordinateur d'accéder à une page et de la lire sur un disque que de traiter ensuite toutes les données lues. C'est pour cette raison que, dans

ce chapitre, nous étudierons séparément les deux composantes principales du temps d'exécution :

- le nombre d'accès au disque et
- le temps CPU (temps de calcul).

Le nombre d'accès au disque se mesure en fonction du nombre de pages de données qui devront être lues ou écrites sur le disque. On remarque que le temps d'accès au disque n'est pas constant : il dépend de la distance entre la piste courante et la piste à atteindre, et aussi de l'état de rotation initial du disque. Nous allons néanmoins utiliser le nombre de pages lues ou écrites comme première approximation du temps total consommé par les accès au disque.

Dans une application de B-arbre classique, la quantité de données gérées est si grande qu'elles ne tiennent pas toutes en même temps dans la mémoire principale. Les algorithmes de B-arbre copient quand il le faut certaines pages du disque vers la mémoire principale et réécrivent sur le disque les pages modifiées. Comme ces algorithmes n'ont besoin que d'un nombre constant de pages en mémoire principale à un instant donné, la taille des B-arbres sur lesquels ils travaillent n'est pas limitée par celle de la mémoire principale.

Les opérations d'accès au disque sont modélisées dans notre pseudo code de la manière suivante. Soit x un pointeur vers un objet. Si l'objet se trouve dans la mémoire principale de l'ordinateur, on peut faire référence aux champs de l'objet de la façon habituelle : $clé[x]$, par exemple. En revanche, si l'objet pointé par x se trouve sur le disque, on doit effectuer l'opération LIRE-DISQUE(x) pour amener l'objet x en mémoire principale avant de pouvoir référencer ses champs. (On suppose que, si x est déjà en mémoire principale, LIRE-DISQUE(x) n'exige pas d'accès disque ; c'est une non-opération ou « nop ».) De la même façon, l'opération ÉCRIRE-DISQUE(x) sert à sauvegarder toutes les modifications intervenues dans les champs de l'objet x . Autrement dit, le modèle général de travail sur un objet est le suivant :

- 1 $x \leftarrow$ un pointeur vers un certain objet
- 2 LIRE-DISQUE(x)
- 3 opérations qui accèdent aux champs de x et/ou les modifient
- 4 ÉCRIRE-DISQUE(x) ▷ Omis si aucun champ de x n'a été modifié.
- 5 autres opérations, qui accèdent aux champs de x sans les modifier

À un instant donné, le système ne peut garder qu'un nombre de pages limité en mémoire principale. On suppose que les pages qui ne sont plus utilisées sont vidées de la mémoire principale par le système ; nos algorithmes de B-arbre ignoreront ce problème.

Puisque dans la plupart des systèmes, le temps d'exécution d'un algorithme de B-arbre est surtout déterminé par le nombre d'opérations LIRE-DISQUE et ÉCRIRE-DISQUE qu'il effectue, il est logique d'utiliser au mieux ces opérations pour qu'elles lisent ou écrivent le plus d'informations possible à chaque accès. Un nœud de B-arbre

aura donc le plus souvent la taille d'une page de disque entière. Le nombre d'enfants que pourra posséder un nœud est donc limité par la taille d'une page sur le disque.

Pour un grand B-arbre stocké sur un disque, on utilise fréquemment des facteurs de ramification compris entre 50 et 2000, qui dépendent du rapport entre la taille d'une clé et la taille d'une page. Un grand facteur de ramification réduit énormément à la fois la hauteur de l'arbre et le nombre d'accès disque nécessaires pour trouver une clé. La figure 18.3 montre un B-arbre ayant un facteur de ramification 1001 et une hauteur 2, capable de stocker plus d'un milliard de clés ; pourtant, comme le nœud situé à la racine peut être gardé en permanence dans la mémoire principale, il suffit de *deux* accès au disque pour accéder à une clé quelconque de cet arbre !

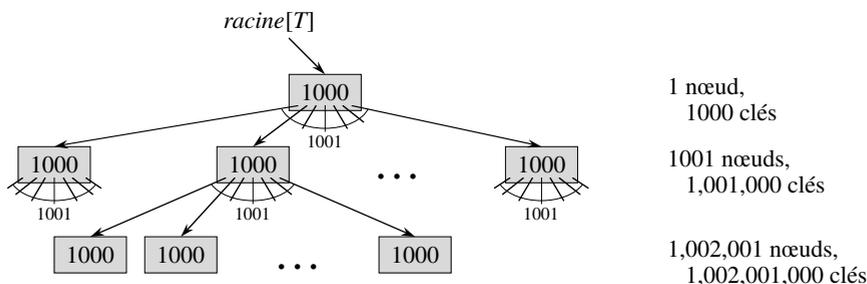


Figure 18.3 Un B-arbre de hauteur 2 contenant plus d'un milliard de clés. Chaque nœud (feuilles comprises) contient 1 000 clés. Il existe 1 001 nœuds à la profondeur 1 et plus d'un millions de feuilles à la profondeur 2. On peut voir à l'intérieur de chaque nœud la valeur $n[x]$, qui représente le nombre de clés de x .

18.1 DÉFINITION D'UN B-ARBRE

Pour simplifier on supposera, comme nous l'avons fait pour les arbres de recherche binaires et les arbres rouge-noir, que toute « donnée satellite » associée à une clé est stockée dans le même nœud que la clé. En pratique, il est possible de ne stocker avec chaque clé qu'un pointeur vers une autre page du disque, qui contiendra les informations satellites associées à cette clé. Le pseudo code de ce chapitre suppose implicitement que les informations satellites (ou le pointeur qui les adresse) voyagent avec la clé à chaque fois que la clé passe d'un nœud à l'autre. Une autre organisation fréquemment utilisée pour les B-arbres consiste à stocker toutes les informations satellites dans les feuilles et à ne conserver dans les nœuds internes que les clés et les pointeurs d'enfant, ce qui permet de maximiser le facteur de ramification des nœuds internes.

Un **B-arbre** T est une arborescence (de racine $racine[T]$) possédant les propriétés suivantes.

1) Chaque nœud x contient les champs ci-dessous :

a) $n[x]$, le nombre de clés conservées actuellement par le nœud x ,

b) les $n[x]$ clés elles-mêmes, stockées par ordre non décroissant :

$$clé_1[x] \leqslant clé_2[x] \leqslant \cdots \leqslant clé_{n[x]}[x],$$

c) *feuille* $[x]$, une valeur booléenne qui vaut VRAI si x est une feuille et FAUX si x est un nœud interne.

2) Chaque nœud interne x contient également $n[x] + 1$ pointeurs $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ vers ses enfants. Les feuilles n'ont pas d'enfants, et leurs champs c_i ne sont donc pas définis.

3) Les clés $clé_i[x]$ déterminent les intervalles de clés stockés dans chaque sous-arbre : si k_i est une clé stockée dans le sous-arbre de racine $c_i[x]$, alors

$$k_1 \leqslant clé_1[x] \leqslant k_2 \leqslant clé_2[x] \leqslant \cdots \leqslant clé_{n[x]}[x] \leqslant k_{n[x]+1}.$$

4) Toutes les feuilles ont la même profondeur, qui est la hauteur h de l'arbre.

5) Il existe un majorant et un minorant pour le nombre de clés pouvant être contenues par un nœud. Ces bornes peuvent être exprimées en fonction d'un entier fixé $t \geqslant 2$, appelé le **degré minimal** du B-arbre :

a) Tout nœud autre que la racine doit contenir au moins $t - 1$ clés. Tout nœud interne autre que la racine possède donc au moins t enfants. Si l'arbre n'est pas vide, la racine doit posséder au moins une clé.

b) Tout nœud peut contenir au plus $2t - 1$ clés. Un nœud interne peut donc posséder au plus $2t$ enfants. On dit qu'un nœud est **complet** s'il contient exactement $2t - 1$ clés. ⁽¹⁾

Le B-arbre le plus simple qui puisse être est celui pour lequel $t = 2$. Tout nœud interne possède alors 2, 3 ou 4 enfants et on se trouve en présence d'un **arbre 2-3-4**. Toutefois, en pratique, on utilise des valeurs de t beaucoup plus grandes.

b) Hauteur d'un B-arbre

Le nombre d'accès disque nécessaires pour la plupart des opérations sur un B-arbre est proportionnel à la hauteur du B-arbre. Analysons à présent la hauteur d'un B-arbre dans le cas le plus défavorable.

Théorème 18.1 Si $n \geqslant 1$, alors, pour tout B-arbre T à n clés de hauteur h et de degré minimal $t \geqslant 2$,

$$h \leqslant \log_t \frac{n+1}{2}.$$

Démonstration : Si un B-arbre a une hauteur h , la racine contient au moins une clé et tous les autres nœuds contiennent au moins $t - 1$ clés. Par conséquent, il y a au moins 2 nœuds à la profondeur 1, au moins $2t$ nœuds à la profondeur 2, au moins $2t^2$

(1) Une autre variante courante de B-arbre, appelée **B*-arbre**, exige que chaque nœud interne soit au moins rempli aux $2/3$, et non pas rempli au moins à moitié comme l'exige un B-arbre.

nœuds à la profondeur 3, etc. jusqu'à la profondeur h il y a au moins $2t^{h-1}$ nœuds. La figure 18.4 montre un tel arbre pour $h = 3$. Ainsi, le nombre n de clés obéit à l'inégalité

$$n \geq 1 + (t - 1) \sum_{i=1}^h 2t^{i-1} = 1 + 2(t - 1) \left(\frac{t^h - 1}{t - 1} \right) = 2t^h - 1 .$$

Avec un peu d'algèbre élémentaire, l'on obtient $t^h \leq (n + 1)/2$. En prenant les logarithmes de base t des deux membres, on démontre le théorème. \square

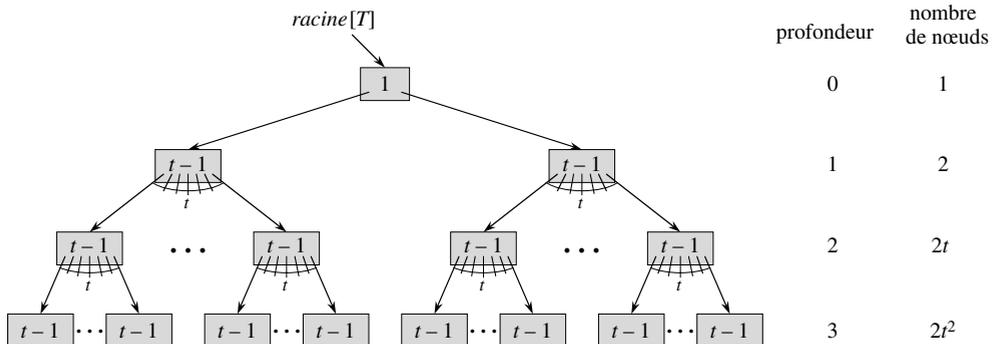


Figure 18.4 B-arbre de hauteur 3 contenant un nombre de clés minimal. À l'intérieur de chaque nœud x est affiché $n[x]$.

On voit ici la puissance des B-arbres comparés aux arbres rouge-noir. Bien que la hauteur de l'arbre augmente en $O(\lg n)$ dans les deux cas (n'oubliez pas que t est une constante), dans le cas des B-arbres la base du logarithme peut être très supérieure. Du coup, les B-arbres économisent un facteur d'environ $\lg t$ par rapport aux arbres rouge-noir, au niveau du nombre de nœuds examinés par la plupart des opérations d'arbre. Comme l'examen d'un nœud demande habituellement un accès disque, le nombre d'accès au disque est sensiblement réduit.

Exercices

18.1.1 Pourquoi n'autorise-t-on pas $t = 1$ comme degré minimal ?

18.1.2 Pour quelles valeurs de t l'arbre de la figure 18.1 est-il un B-arbre valide ?

18.1.3 Donner tous les B-arbres valides de degré minimal 2 permettant de représenter $\{1, 2, 3, 4, 5\}$.

18.1.4 En tant que fonction du degré minimal t , quel est le nombre maximal de clés qui peuvent être stockées dans un B-arbre de hauteur h ?

18.1.5 Décrire la structure de données que l'on obtiendrait si chaque nœud noir d'un arbre rouge-noir devait absorber ses enfants rouges, en incorporant leurs enfants aux siens.

18.2 OPÉRATIONS FONDAMENTALES SUR LES B-ARBRES

Dans cette section, nous présentons en détail les opérations RECHERCHER-B-ARBRE, CRÉER-B-ARBRE et INSÉRER-B-ARBRE. Pour ces procédures, nous adopterons deux conventions :

- La racine du B-arbre se trouve toujours en mémoire principale, de manière qu’une opération LIRE-DISQUE ne soit jamais nécessaire pour la racine ; cependant, un appel à ÉCRIRE-DISQUE sera nécessaire à chaque modification du nœud racine.
- Tout nœud passé en paramètre devra avoir subi auparavant une opération LIRE-DISQUE.

Les procédures présentées ici implémentent toutes des algorithmes « une passe » qui partent de la racine pour descendre l’arbre mais qui ne reviennent jamais en arrière.

a) Recherche dans un B-arbre

Les recherches dans un B-arbre ressemblent beaucoup à celles effectuées dans un arbre binaire de recherche, excepté qu’au lieu de prendre à chaque nœud une décision de branchement binaire (deux options au choix), on prend une décision à options multiples, selon le nombre d’enfants du nœud. Plus précisément, à chaque nœud interne x , on prend une décision parmi $(n[x] + 1)$ options possibles.

RECHERCHER-B-ARBRE est une généralisation directe de la procédure RECHERCHER-ARBRE définie pour les arbres binaires de recherche. RECHERCHER-B-ARBRE prend en entrée un pointeur vers la racine x d’un sous-arbre et une clé k à rechercher dans ce sous-arbre. L’appel de niveau supérieur est donc de la forme RECHERCHER-B-ARBRE(*racine*[T], k). Si k se trouve dans le B-arbre, RECHERCHER-B-ARBRE retourne le couple (y, i) constitué d’un nœud y et d’un indice i tel que $clé_i[y] = k$. Sinon, la valeur NIL est retournée.

```

RECHERCHER-B-ARBRE( $x, k$ )
1   $i \leftarrow 1$ 
2  tant que  $i \leq n[x]$  et  $k > clé_i[x]$ 
3    faire  $i \leftarrow i + 1$ 
4  si  $i \leq n[x]$  et  $k = clé_i[x]$ 
5    alors retourner  $(x, i)$ 
6  si feuille[ $x$ ]
7    alors retourner NIL
8  sinon LIRE-DISQUE( $c_i[x]$ )
9    retourner RECHERCHER-B-ARBRE( $c_i[x], k$ )

```

En employant une méthode de recherche linéaire, les lignes 1–3 trouvent le plus petit indice i pour lequel $k \leq clé_i[x]$ ou, s’il n’est pas trouvé, donnent à i la valeur $n[x] + 1$. Les lignes 4–5 testent si la clé a été découverte, auquel cas elle est retournée. Les lignes 6–9 permettent soit d’arrêter la recherche si elle échoue (si x est une feuille),

soit d'effectuer une recherche récursive dans le sous-arbre de x approprié, après avoir effectué sur ce enfant le LIRE-DISQUE obligatoire.

La figure 18.1 illustre l'action de RECHERCHER-B-ARBRE ; les nœuds en gris clair sont ceux examinés pendant la recherche de la clé R .

Comme pour la procédure RECHERCHER-ARBRE des arbres binaires de recherche, les nœuds rencontrés pendant la récursivité forment un chemin qui descend de la racine de l'arbre. Le nombre de pages disque auxquelles RECHERCHER-B-ARBRE accède est donc $O(h) = O(\log_t n)$, où h est la hauteur du B-arbre et n le nombre de clés qu'il contient. Comme $n[x] < 2t$, le temps pris par la boucle **tant que** des lignes 2–3 à l'intérieur de chaque nœud est $O(t)$ et le temps CPU total est $O(th) = O(t \log_t n)$.

b) Création d'un B-arbre vide

Pour construire un B-arbre T , on commence par appeler CRÉER-B-ARBRE pour créer un nœud racine vide ; ensuite, on appelle INSÉRER-B-ARBRE pour ajouter de nouvelles clés. Ces deux procédures utilisent une procédure auxiliaire ALLOUER-NŒUD, qui alloue en $O(1)$ une page disque pour le nouveau nœud. On peut supposer qu'un nœud créé par ALLOUER-NŒUD ne nécessite aucun appel à LIRE-DISQUE, puisqu'il n'existe pour l'instant aucune information utile sur le disque concernant ce nœud.

CRÉER-B-ARBRE(T)

- 1 $x \leftarrow$ ALLOUER-NŒUD()
- 2 $feuille[x] \leftarrow$ VRAI
- 3 $n[x] \leftarrow 0$
- 4 ÉCRIRE-DISQUE(x)
- 5 $racine[T] \leftarrow x$

CRÉER-B-ARBRE requiert $O(1)$ opérations de disque, pour un temps CPU $O(1)$.

c) Insertion d'une clé dans un B-arbre

L'insertion d'une clé dans un B-arbre est nettement plus complexe que l'insertion d'une clé dans un arbre binaire de recherche. Comme c'est le cas avec un arbre binaire de recherche, on cherche la position de feuille à laquelle il faut insérer la nouvelle clé. Mais avec un B-arbre, on ne peut pas se contenter de créer une nouvelle feuille puis de l'insérer, car l'arbre résultant ne serait pas un B-arbre licite. À la place, on insère la nouvelle clé dans un nœud feuille existant. Comme on ne peut pas insérer une clé dans un nœud feuille qui est plein, on introduit une opération qui *partage* un nœud y plein (ayant $2t - 1$ clés) autour de sa *clé médiane* $clé_t[y]$, pour en faire deux nœuds ayant chacun $t - 1$ clés. La clé médiane remonte dans le parent de y pour identifier le point de partage entre les deux nouveaux arbres. Mais si le parent de y est plein, lui aussi, il faut le partager avant d'y insérer la nouvelle clé ; par conséquent, il se peut que ce processus de partage de nœuds pleins se propage dans tout l'arbre (en partant du bas pour aller vers le haut).

Comme c'est le cas avec un arbre binaire de recherche, on peut insérer une clé dans un B-arbre en une seule passe qui descend l'arbre depuis la racine pour arriver sur une feuille. Pour ce faire, on n'attend pas de savoir s'il faudra partager un nœud plein pour faire l'insertion. À la place, à mesure que l'on descend l'arbre pour chercher la position de la nouvelle clé, on partage chaque nœud plein que l'on rencontre sur le chemin (y compris la feuille elle-même). Ainsi, quand on voudra partager un nœud plein y , on sera assuré que son parent n'est pas plein.

► Partage de nœud dans un B-arbre

La procédure PARTAGER-ENFANT-B-ARBRE prend en entrée un nœud interne *non plein* x (supposé être en mémoire centrale), un indice i et un nœud y (censé, lui aussi, être en mémoire centrale), tels que $y = c_i[x]$ soit un enfant *plein* de x . La procédure partage alors cet enfant en deux et modifie x pour qu'il ait un enfant de plus. (Pour partager une racine pleine, on commence par faire de la racine un enfant d'un nouveau nœud racine vide de façon à pouvoir utiliser PARTAGER-ENFANT-B-ARBRE). La hauteur de l'arbre augmente donc de un ; le partage est la seule façon de faire croître l'arbre.

La figure 18.5 illustre ce processus. Le nœud plein y est partagé au niveau de sa clé médiane S , qui remonte vers le parent de y , à savoir le nœud x . Les clés de y qui sont supérieures à la clé médiane vont dans un nouveau nœud z , qui devient un nouvel enfant de x .

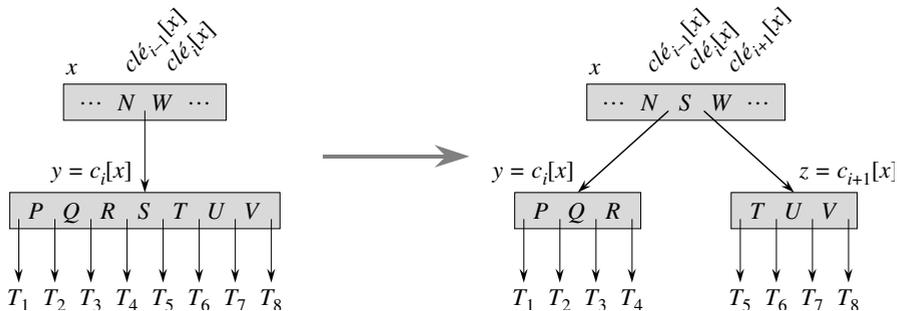


Figure 18.5 Partage d'un nœud avec $t = 4$. Le nœud y est partagé en deux nœuds, y et z , et la clé médiane S de y va dans le parent de y .

PARTAGER-ENFANT-B-ARBRE(x, i, y)

- 1 $z \leftarrow \text{ALLOUER-NŒUD}()$
- 2 $\text{feuille}[z] \leftarrow \text{feuille}[y]$
- 3 $n[z] \leftarrow t - 1$
- 4 **pour** $j \leftarrow 1$ **à** $t - 1$
- 5 **faire** $\text{clé}_j[z] \leftarrow \text{clé}_{j+t}[y]$
- 6 **si non** $\text{feuille}[y]$
- 7 **alors pour** $j \leftarrow 1$ **à** t
- 8 **faire** $c_j[z] \leftarrow c_{j+t}[y]$

```

9   $n[y] \leftarrow t - 1$ 
10 pour  $j \leftarrow n[x] + 1$  jusqu'à  $i + 1$ 
11     faire  $c_{j+1}[x] \leftarrow c_j[x]$ 
12      $c_{i+1}[x] \leftarrow z$ 
13 pour  $j \leftarrow n[x]$  jusqu'à  $i$ 
14     faire  $clé_{j+1}[x] \leftarrow clé_j[x]$ 
15      $clé_i[x] \leftarrow clé_i[y]$ 
16      $n[x] \leftarrow n[x] + 1$ 
17 ÉCRIRE-DISQUE( $y$ )
18 ÉCRIRE-DISQUE( $z$ )
19 ÉCRIRE-DISQUE( $x$ )

```

PARTAGER-ENFANT-B-ARBRE fonctionne en faisant du « couper-coller » direct. Ici, y est le i ème enfant de x et c 'est le nœud qui est partagé. Le nœud y a, à l'origine, $2t$ enfants ($2t - 1$ clés), mais il est réduit à t enfants ($t - 1$ clés) par cette opération. Le nœud z « adopte » les t plus grands enfants ($t - 1$ clés) de y , et z devient un nouvel enfant de x , placé juste après y dans la table des enfants de x . La clé médiane de y remonte pour devenir la clé de x qui sépare y et z .

Les lignes 1–8 créent le nœud z , puis lui donnent les $t - 1$ plus grandes clés et les t enfants correspondants de y . La ligne 9 modifie le compteur de clés de y . Enfin, les lignes 10–16 insèrent z en tant qu'enfant de x , font remonter la clé médiane de y à x afin de séparer y de z , puis ajustent le compteur de clés de x . Les lignes 17–19 réécrivent sur disque toutes les pages modifiées. Le temps CPU consommé par PARTAGER-ENFANT-B-ARBRE est $\Theta(t)$, en raison des boucles des lignes 4–5 et 7–8. (Les autres boucles font $O(t)$ itérations.) La procédure effectue $O(1)$ opérations sur le disque.

► Insertion d'une clé dans un B-arbre en une seule passe descendante

On insère une clé k dans un B-arbre T de hauteur h en une seule passe qui descend l'arbre, au prix de $O(h)$ accès disque. Le temps CPU requis est $O(th) = O(t \log n)$. La procédure INSÉRER-B-ARBRE emploie PARTAGER-ENFANT-B-ARBRE pour assurer que la récursivité ne descendra jamais sur un nœud plein.

```

INSÉRER-B-ARBRE( $T, k$ )
1   $r \leftarrow racine[T]$ 
2  si  $n[r] = 2t - 1$ 
3     alors  $s \leftarrow ALLOUER-NŒUD()$ 
4          $racine[T] \leftarrow s$ 
5          $feuille[s] \leftarrow \text{FAUX}$ 
6          $n[s] \leftarrow 0$ 
7          $c_1[s] \leftarrow r$ 
8         PARTAGER-ENFANT-B-ARBRE( $s, 1, r$ )
9         INSÉRER-B-ARBRE-INCOMPLET( $s, k$ )
10 sinon INSÉRER-B-ARBRE-INCOMPLET( $r, k$ )

```

Les lignes 3–9 gèrent le cas où le nœud racine r est plein : la racine est alors partagée, et un nouveau nœud s (ayant deux enfants) devient la racine. Partager la racine est l'unique façon d'augmenter la hauteur d'un B-arbre. La figure 18.6 illustre ce cas. Contrairement à un arbre binaire de recherche, un B-arbre croît par le haut et non par le bas. La procédure s'achève par l'appel de `INSÉRER-B-ARBRE-INCOMPLET` pour faire l'insertion de la clé k dans l'arbre enraciné sur le nœud racine non plein. `INSÉRER-B-ARBRE-INCOMPLET` fait autant d'appels récursifs que nécessaire au cours de sa descente de l'arbre, garantissant à tout instant que le nœud sur lequel elle fait de la récursivité n'est pas plein, et ce en appelant `PARTAGER-ENFANT-B-ARBRE` en fonction des besoins.

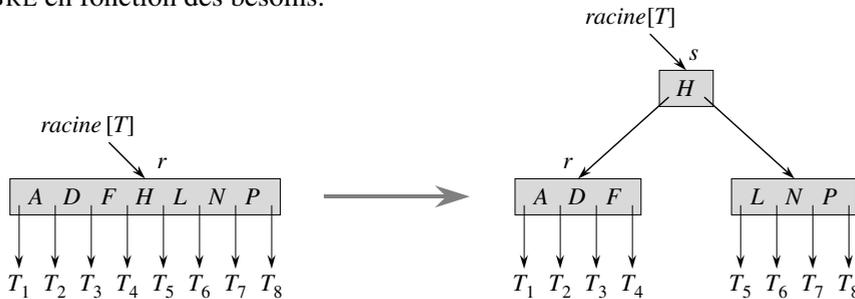


Figure 18.6 Partage de la racine avec $t = 4$. Le nœud racine r est partagé en deux, et il y a création d'un nouveau nœud racine s . La nouvelle racine contient la clé médiane de r et a comme enfants les deux moitiés de r . Le B-arbre croît d'une unité quand il y a un partage de la racine.

La procédure auxiliaire récursive `INSÉRER-B-ARBRE-INCOMPLET` insère les clés k dans le nœud x , qui est censé être non plein lors de l'appel de la procédure. Le fonctionnement de `INSÉRER-B-ARBRE` et le fonctionnement récursif de `INSÉRER-B-ARBRE-INCOMPLET` assurent que cette hypothèse est vraie.

`INSÉRER-B-ARBRE-INCOMPLET`(x, k)

```

1   $i \leftarrow n[x]$ 
2  si feuille[ $x$ ]
3      alors tant que  $i \geq 1$  et  $k < clé_i[x]$ 
4          faire  $clé_{i+1}[x] \leftarrow clé_i[x]$ 
5               $i \leftarrow i - 1$ 
6           $clé_{i+1}[x] \leftarrow k$ 
7           $n[x] \leftarrow n[x] + 1$ 
8          ÉCRIRE-DISQUE( $x$ )
9      sinon tant que  $i \geq 1$  et  $k < clé_i[x]$ 
10         faire  $i \leftarrow i - 1$ 
11          $i \leftarrow i + 1$ 
12         LIRE-DISQUE( $c_i[x]$ )
13         si  $n[c_i[x]] = 2t - 1$ 
14             alors PARTAGER-ENFANT-B-ARBRE( $x, i, c_i[x]$ )
15                 si  $k > clé_i[x]$ 
16                     alors  $i \leftarrow i + 1$ 
17         INSÉRER-B-ARBRE-INCOMPLET( $c_i[x], k$ )

```

La procédure INSÉRER-B-ARBRE-INCOMPLET fonctionne ainsi. Les lignes 3–8 gèrent le cas où x est une feuille, en insérant la clé k dans x . Si x n'est pas une feuille, alors on doit insérer k dans le nœud feuille idoïne du sous-arbre enraciné sur le nœud interne x . Dans ce cas, les lignes 9–11 déterminent l'enfant de x sur lequel doit descendre la récursivité. La ligne 13 détecte si la récursivité descendrait sur un enfant plein, auquel cas la ligne 14 utilise PARTAGER-ENFANT-B-ARBRE pour diviser cet enfant en deux enfants non pleins, et les lignes 15–16 déterminent lequel des deux enfants est maintenant l'enfant sur lequel il faut descendre. (Notez que point n'est besoin d'un appel LIRE-DISQUE($c_i[x]$) après que la ligne 16 a incrémenté i , car la récursivité descendra dans ce cas sur un enfant qui vient d'être créé par PARTAGER-ENFANT-B-ARBRE.) L'effet net des lignes 13–16 est donc de garantir que la procédure ne va jamais faire de récursivité sur un nœud plein. La ligne 17 fait ensuite de la récursivité pour insérer k dans le sous-arbre idoïne. La figure 18.7 illustre les divers cas d'insertion dans un B-arbre.

Le nombre d'accès disque effectué par INSÉRER-B-ARBRE est $O(h)$ pour un B-arbre de hauteur h , car il n'y a que $O(1)$ opérations LIRE-DISQUE et ÉCRIRE-DISQUE qui sont faites entre les appels à INSÉRER-B-ARBRE-INCOMPLET. Le temps CPU total consommé est de $O(th) = O(t \log_t n)$. Comme INSÉRER-B-ARBRE-INCOMPLET est récursive par la queue, on peut aussi la mettre en œuvre sous la forme d'une boucle **tant que**, démontrant que le nombre de pages qui doivent résider en mémoire centrale à un instant quelconque est $O(1)$.

Exercices

18.2.1 Montrer les résultats de l'insertion successive des clés

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

dans un B-arbre vide de degré minimal 2. Ne représenter que les configurations de l'arbre qui précèdent immédiatement le découpage d'un nœud, puis représenter la configuration finale.

18.2.2 Décrire dans quelles circonstances éventuelles l'exécution d'un appel INSÉRER-B-ARBRE pourrait engendrer des opérations LIRE-DISQUE ou ÉCRIRE-DISQUE redondantes. (Un LIRE-DISQUE redondant concerne une page qui se trouve déjà en mémoire. Un ÉCRIRE-DISQUE redondant écrit sur le disque une page identique à celle déjà présente sur le disque.)

18.2.3 Dire comment trouver la clé minimale stockée dans un B-arbre et comment trouver le prédécesseur d'une clé donnée d'un B-arbre.

18.2.4 * On suppose que les clés $\{1, 2, \dots, n\}$ sont insérées dans un B-arbre vide de degré minimal 2. Combien de nœuds possédera le B-arbre final ?

18.2.5 Comme les feuilles n'ont pas besoin de pointeurs d'enfant, on pourrait concevoir qu'elles utilisent une valeur de t différente (plus grande) que les nœuds internes pour la même taille de page disque. Montrer comment modifier les procédures de création et d'insertion dans un B-arbre pour gérer cette variante.

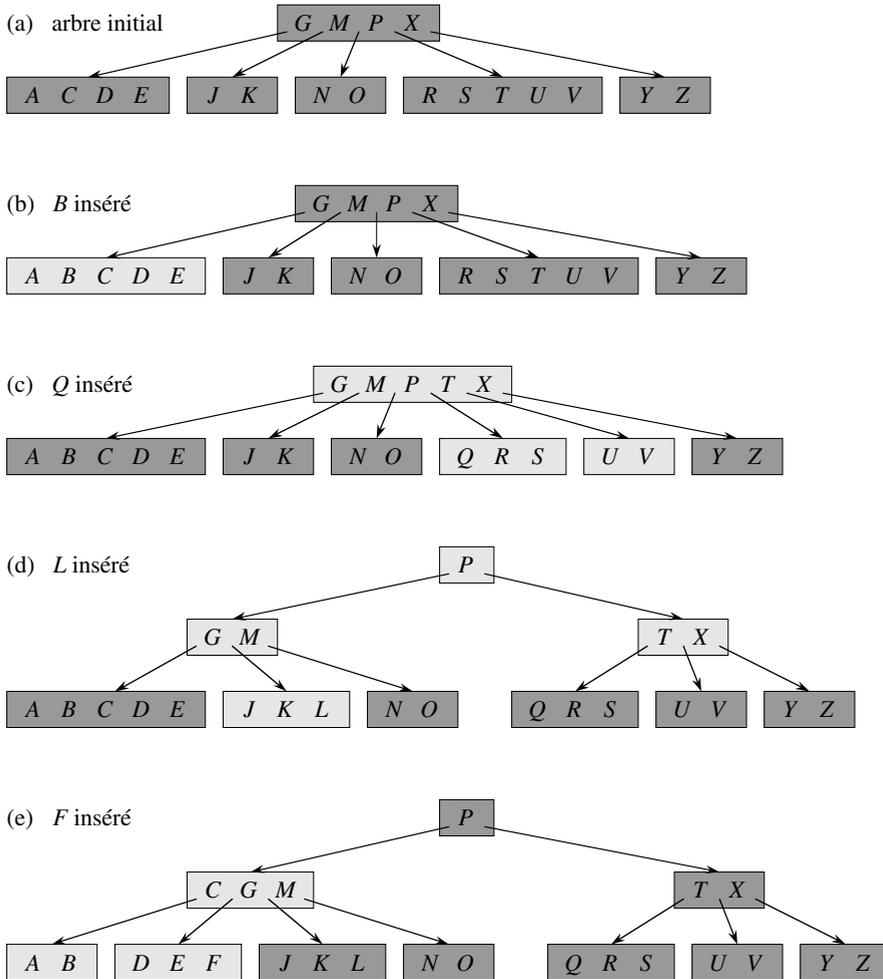


Figure 18.7 Insertion de clés dans un B-arbre. Le degré minimal t de ce B-arbre est 3, de sorte qu'un nœud peut contenir au plus 5 clés. Les nœuds en gris clair sont ceux qui seront modifiés par le processus d'insertion. (a) L'arbre initial de cet exemple. (b) Le résultat de l'insertion de *B* dans l'arbre initial; c'est une insertion simple dans une feuille. (c) Le résultat de l'insertion de *Q* dans l'arbre précédent. Le nœud *RSTUV* est divisé en deux nœuds contenant *RS* et *UV*, la clé *T* remonte dans la racine et *Q* est inséré dans la moitié de gauche (nœud *RS*). (d) Le résultat de l'insertion de *L* dans l'arbre précédent. La racine est divisée immédiatement car elle est pleine, et le B-arbre croît en hauteur de un. Ensuite, *L* est inséré dans la feuille contenant *JK*. (e) Le résultat de l'insertion de *F* dans l'arbre précédent. Le nœud *ABCDE* est divisé, puis *F* est inséré dans la moitié de droite (nœud *DE*).

18.2.6 On suppose que RECHERCHER-B-ARBRE est implémentée de manière à utiliser une recherche dichotomique, plutôt qu'une recherche linéaire, à l'intérieur de chaque nœud. Montrer que le temps CPU requis devient alors $O(\lg n)$, indépendamment de la façon dont t est choisi comme fonction de n .

18.2.7 On suppose que le disque est construit de façon qu'on puisse choisir arbitrairement la taille d'une page disque, mais que le temps pris pour lire la page soit $a + bt$, où a et b sont des constantes spécifiées et t le degré minimal d'un B-arbre utilisant des pages de cette taille. Décrire la manière de choisir t pour minimiser (approximativement) le temps de recherche dans le B-arbre. Suggérer une valeur optimale de t pour le cas où $a = 5$ ms et $b = 10$ microsecondes.

18.3 SUPPRESSION D'UNE CLÉ DANS UN B-ARBRE

La suppression dans un B-arbre ressemble à l'insertion, en un peu plus compliqué. En effet, la suppression d'une clé peut concerner un nœud quelconque et pas seulement une feuille ; en outre, la suppression dans un nœud interne exige que les enfants du nœud soient réorganisés. Comme avec l'insertion, il faut éviter qu'une suppression ne produise un arbre dont la structure enfreint les propriétés de B-arbre. De même que nous devons faire en sorte qu'un nœud ne grossisse pas trop pour cause d'insertion, nous devons faire en sorte qu'un nœud ne rétrécisse pas trop pour cause de suppression (sachant que la racine est autorisée à avoir moins de clés que le minimum $t - 1$, mais qu'elle n'a pas le droit d'en avoir plus que le maximum $2t - 1$). Un algorithme d'insertion simpliste risquerait d'être obligé de rebrousser chemin si le nœud où il faudrait insérer la clé est plein ; de même, une approche simpliste en matière de suppression nous obligerait à revenir en arrière si le nœud (autre que la racine) dans lequel il faudrait supprimer la clé a le nombre minimal de clés.

Supposons que la procédure SUPPRIMER-B-ARBRE doive supprimer la clé k du sous-arbre enraciné en x . Cette procédure est faite de telle façon que, chaque fois que SUPPRIMER-B-ARBRE est appelée récursivement sur un nœud x , le nombre de clés de x est au moins égal au degré minimal t . Notez que cette condition exige une clé de plus que le minimum requis par les conditions de B-arbre ; il peut donc advenir qu'une clé soit obligée d'aller dans un nœud enfant avant que la récursivité descende sur cet enfant. Cette condition renforcée permet de supprimer une clé de l'arbre en une seule passe descendante, sans risque de « retour en arrière » (à une seule exception, que nous allons expliquer). À propos des spécifications suivantes concernant la suppression dans un B-arbre, il faut bien comprendre ceci : s'il arrive que le nœud racine x devienne un nœud interne n'ayant pas de clés (cette situation peut se produire dans les cas 2c et 3b, donnés ci-après), alors x est supprimé et l'unique enfant de x , à savoir $c_1[x]$, devient la nouvelle racine de l'arbre (dont la hauteur diminue de un) et préserve la propriété énonçant que la racine de l'arbre contient au moins une clé (sauf si l'arbre est vide).

Nous allons esquisser le fonctionnement de la suppression, au lieu de présenter le pseudo code. La figure 18.8 illustre les divers cas de la suppression de clé dans un B-arbre.

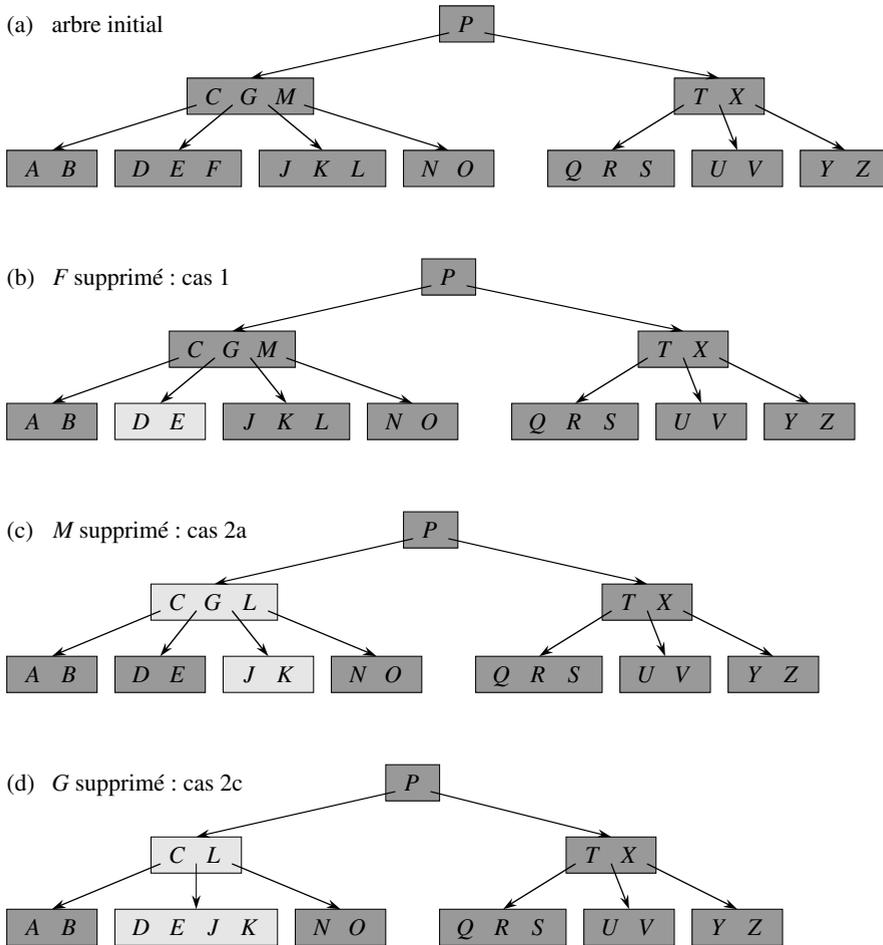
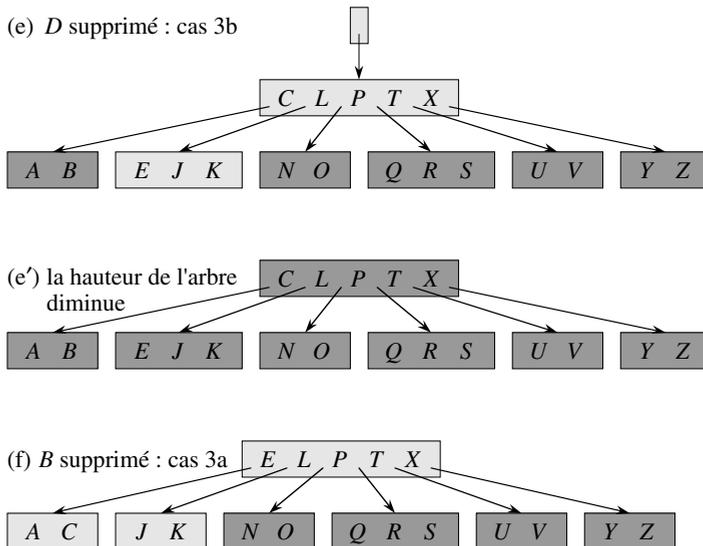


Figure 18.8 Suppression de clés dans un B-arbre. Le degré minimal de ce B-arbre est $t = 3$, de sorte qu'un nœud (autre que la racine) ne peut pas avoir moins de 2 clés. Les nœuds qui sont modifiés sont en gris clair. (a) Le B-arbre de la figure 18.7(e). (b) Suppression de F . C'est le cas 1 : suppression simple dans une feuille. (c) Suppression de M . C'est le cas 2a : le prédécesseur L de M remonte pour prendre la place de M . (d) Suppression de G . C'est le cas 2c : G est poussé vers le bas pour créer le nœud $DEGJK$, puis G est supprimé de cette feuille (cas 1). (e) Suppression de D . C'est le cas 3b : la récursivité ne peut pas descendre sur le nœud CL , car il n'a que 2 clés; donc, P est poussé vers le bas et fusionné avec CL et TX pour former $CLPTX$; ensuite, D est supprimé d'une feuille (cas 1). (e') Après (d), la racine est supprimée et la hauteur de l'arbre diminue de un. (f) Suppression de B . C'est le cas 3a : C est déplacé pour occuper la place de B et E est déplacé pour occuper la place de C .



- 1) Si la clé k est dans le nœud x et que x est une feuille, on supprime la clé k de x .
- 2) Si la clé k est dans le nœud x et que x est un nœud interne, on procède comme suit.
 - a) Si l'enfant y qui précède k dans le nœud x a au moins t clés, on cherche le prédécesseur k' de k dans le sous-arbre enraciné en y . On supprime récursivement k' , puis on remplace k par k' dans x . (Trouver k' et le supprimer sont deux actions qui peuvent se faire dans une même passe descendante.)
 - b) De manière symétrique, si l'enfant z qui suit k dans le nœud x a au moins t clés, on cherche le successeur k' de k dans le sous-arbre enraciné en z . On supprime récursivement k' , puis on remplace k par k' dans x . (Trouver k' et le supprimer sont deux actions qui peuvent se faire dans une même passe descendante.)
 - c) Sinon, si y et z n'ont que $t - 1$ clés chacun, on fusionne k avec le contenu de z et on fait passer le tout dans y ; ce faisant, x perd k plus le pointeur vers z et y contient désormais $2t - 1$ clés. Ensuite, on libère z et l'on supprime récursivement k de y .
- 3) Si la clé k ne figure pas dans le nœud interne x , on cherche la racine $c_i[x]$ du sous-arbre contenant k (pour autant que k soit dans l'arbre). Si $c_i[x]$ n'a que $t - 1$ clés, on exécute l'étape 3a ou 3b (selon les besoins) pour garantir que l'on va descendre sur un nœud qui contient bien au moins t clés. Ensuite, on termine en appliquant la récursivité à l'enfant approprié de x .
 - a) Si $c_i[x]$ n'a que $t - 1$ clés mais qu'il a un frère immédiat ayant au moins t clés, on donne à $c_i[x]$ une clé supplémentaire de la façon que voici : on prend une clé de x que l'on fait descendre dans $c_i[x]$, on prend une clé du frère immédiat (de gauche ou de droite) de $c_i[x]$ que l'on fait monter dans x , et on déplace le pointeur d'enfant approprié pour le faire passer du frère à $c_i[x]$.

- b) Si les frères immédiats de $c_i[x]$ et de $c_i[x]$'s ont chacun $t - 1$ clés, on fusionne $c_i[x]$ avec un frère ; cela implique qu'il faut faire descendre une clé de x dans le nœud nouvellement fusionné pour qu'elle devienne la clé médiane de ce nœud.

Comme la plupart des clés d'un B-arbre sont dans les feuilles, on peut s'attendre à ce que, dans la pratique, les suppressions de clé se fassent le plus souvent sur des feuilles. La procédure SUPPRIMER-B-ARBRE travaille alors en une seule passe descendante, sans risque de devoir revenir en arrière dans l'arbre. En revanche, quand on supprime une clé d'un nœud interne, la procédure descend l'arbre mais elle peut éventuellement être amenée à revenir sur le nœud dont on a supprimé la clé, pour remplacer la clé par son prédécesseur ou successeur (cas 2a et 2b).

Cette procédure peut paraître compliquée, mais elle ne demande que $O(h)$ accès disque pour un B-arbre de hauteur h ; en effet, il n'y a que $O(1)$ appels LIRE-DISQUE et ÉCRIRE-DISQUE entre les invocations récursives de la procédure. Le temps CPU requis est $O(th) = O(t \log_t n)$.

Exercices

18.3.1 Montrer le résultat de la suppression dans l'ordre de C , P et V dans l'arbre de la figure 18.8(f).

18.3.2 Écrire le pseudo code de SUPPRIMER-B-ARBRE.

PROBLÈMES

18.1. Piles sur un espace de stockage secondaire

On considère l'implémentation d'une pile dans un ordinateur assez limité en mémoire principale et bien pourvu en espace disque. Les opérations EMPILER et DÉPILER sont supportées pour des valeurs contenues dans un mot. La pile devra pouvoir croître jusqu'à dépasser largement la mémoire principale disponible, ce qui impose d'en stocker la plus grande partie sur disque.

Conserver la pile entièrement sur le disque est simple quoique peu efficace. On garde en mémoire un pointeur vers la pile, qui représente l'adresse sur disque du sommet de la pile. Si le pointeur a une valeur p , l'élément du sommet est le $(p \bmod m)$ ème mot sur la page $\lfloor p/m \rfloor$ du disque, où m est le nombre de mots par page.

Pour implémenter l'opération EMPILER, on incrémente le pointeur de pile, on charge en mémoire la page disque concernée, on copie l'élément à empiler dans le mot approprié sur la page et on écrit la page ainsi modifiée sur le disque. L'opération DÉPILER est similaire. On décrémente le pointeur de pile, on charge la page concernée à partir du disque et on retourne le sommet de la pile. Il est inutile de réécrire la page sur le disque puisqu'elle n'est pas modifiée par l'opération.

Les opérations de disque étant relativement coûteuses, on compte deux choses pour toute implémentation : le nombre total d'accès disque et le temps processeur total. Un accès disque à une page de m mots compte pour un accès disque et pour un temps CPU de $\Theta(m)$.

- a. Asymptotiquement, quel est le nombre d'accès au disque dans le cas le plus défavorable pour n opérations de pile ayant recours à cette implémentation naïve ? Quel est le temps CPU pris par les n opérations de pile ? (Exprimer les résultats en fonction de m et n . Idem pour les questions suivantes.)

On considère à présent une implémentation pour laquelle une page de la pile est conservée en mémoire. (On réserve également une petite quantité de mémoire pour savoir quelle page se trouve en mémoire à un instant donné.) Une opération de pile ne peut s'effectuer que si la page disque concernée réside en mémoire. Si besoin est, la page se trouvant en mémoire peut être sauvegardée sur le disque et une nouvelle page peut être chargée à partir du disque. Si la page disque se trouve déjà en mémoire, aucun accès au disque n'est nécessaire.

- b. Quel est, dans le cas le plus défavorable, le nombre d'accès au disque requis pour n opérations EMPILER ? Quel est le temps CPU requis ?
- c. Quel est, dans le cas le plus défavorable, le nombre d'accès au disque requis pour n opérations de pile ? Quel est le temps CPU requis ?

Supposons maintenant que la pile soit implémentée en conservant deux pages en mémoire (en plus d'un petit nombre de mots pour la gestion interne).

- d. Décrire comment gérer les pages de pile pour que le nombre amorti d'accès au disque pour toute opération de pile soit $O(1/m)$ et le temps CPU amorti pour toute opération de pile soit $O(1)$.

18.2. Jointure et scission d'arbres 2-3-4

L'opération *jointure* prend deux ensembles dynamiques S' et S'' et un élément x tel que, pour tout $x' \in S'$ et $x'' \in S''$, on ait $clé[x'] < clé[x] < clé[x'']$. Elle retourne un ensemble $S = S' \cup \{x\} \cup S''$. L'opération *scission* est un peu « l'inverse » de la jointure : étant donné un ensemble dynamique S et un élément $x \in S$, elle crée un ensemble S' constitué de tous les éléments de $S - \{x\}$ dont les clés sont inférieures à $clé[x]$ et un ensemble S'' constitué de tous les éléments de $S - \{x\}$ dont les clés sont supérieures à $clé[x]$. Dans ce problème, on cherche à savoir comment implémenter ces opérations sur des arbres 2-3-4. On suppose par commodité que les éléments ne contiennent que des clés et que toutes les valeurs de clé sont distinctes.

- a. Montrer comment gérer, pour tout nœud x d'un arbre 2-3-4, la hauteur du sous-arbre enraciné en x dans un champ $hauteur[x]$. S'assurer que l'implémentation retenue n'affecte pas les temps d'exécution asymptotiques de la recherche, de l'insertion et de la suppression.

- b. Montrer comment implémenter l'opération de jointure. Étant donnés deux arbres 2-3-4 T' et T'' et une clé k , la jointure devra s'exécuter en $O(1 + |h' - h''|)$, où h' et h'' sont les hauteurs respectives de T' et T'' .
- c. Soit T un arbre 2-3-4. Soient p le chemin reliant la racine de T à une clé donnée k , S' l'ensemble des clés de T inférieures à k et S'' l'ensemble des clés de T supérieures à k . Montrer que p partage S' en un ensemble d'arbres $\{T'_0, T'_1, \dots, T'_m\}$ et un ensemble de clés $\{k'_1, k'_2, \dots, k'_m\}$, où, pour $i = 1, 2, \dots, m$, on a $y < k'_i < z$ pour toutes clés $y \in T'_{i-1}$ et $z \in T'_i$. Quelle est la relation entre les hauteurs de T'_{i-1} et T'_i ? Décrire la façon dont p partage S'' en deux ensembles d'arbres et de clés.
- d. Montrer comment implémenter l'opération de scission sur T . Utiliser l'opération de jointure pour rassembler les clés de S' dans un même arbre 2-3-4 T' et les clés de S'' dans un même arbre 2-3-4 T'' . Le temps d'exécution de l'opération de scission devra être $O(\lg n)$, où n est le nombre de clés de T . (*Conseil* : Les coûts des jointures doivent s'annuler mutuellement en partie.)

NOTES

Knuth [185], Aho, Hopcroft et Ullman [5], et Sedgewick [269] fournissent des études plus détaillées sur les arbres équilibrés et les B-arbres. Comer [66] couvre de façon exhaustive les B-arbres. Guibas et Sedgewick [135] étudient les relations entre les différentes sortes d'arbres équilibrés, notamment les arbres rouge-noir et les arbres 2-3-4.

En 1970, J. E. Hopcroft inventa les arbres 2-3, précurseurs des B-arbres et des arbres 2-3-4, dans lesquels tout nœud interne a deux ou trois enfant. Les B-arbres ont été introduits par Bayer et McCreight en 1972 [32] ; ils n'expliquent pas le choix de ce nom.

Bender, Demaine et Farach-Colton [37] ont étudié les performances de B-arbres dans un contexte de gestion hiérarchisée de la mémoire. Leurs algorithmes *à transparence de cache* donnent de bonnes performances sans se soucier explicitement de la taille des données qui sont transférées entre les divers niveaux de la hiérarchie des mémoires.

Chapitre 19

Tas binomiaux

Ce chapitre et le chapitre 20 abordent des structures de données connues sous le nom de *tas fusionnables*, qui supportent les cinq opérations suivantes.

CRÉER-TAS() crée et retourne un nouveau tas sans élément.

INSÉRER(T, x) insère dans le tas T un nœud x , dont le champ *clé* a déjà été rempli.

MINIMUM(T) retourne un pointeur sur le nœud dont la clé est minimale dans le tas T .

EXTRAIRE-MIN(T) supprime du tas T le nœud dont la clé est minimale et retourne un pointeur sur ce nœud.

UNION(T_1, T_2) crée et retourne un nouveau tas qui contient tous les nœuds des tas T_1 et T_2 . Les tas T_1 et T_2 sont « détruits » par cette opération.

Les structures de données utilisées dans ces chapitres supportent aussi les deux opérations suivantes.

DIMINUER-CLÉ(T, x, k) affecte au nœud x du tas T la nouvelle valeur de clé k , qui est censée être inférieure ou égale à la valeur courante de la clé. ⁽¹⁾

SUPPRIMER(T, x) supprime le nœud x du tas T .

Comme le montre le tableau de la figure 19.1, si l'opération UNION n'est pas nécessaire, les tas binaires ordinaires, comme ceux utilisés pour le tri par tas (Chapitre 6), sont suffisants. Les opérations autres que UNION s'exécutent au plus en $O(\lg n)$ dans le cas le plus défavorable sur un tas binaire. Toutefois, quand l'opération UNION

(1) Comme mentionné dans l'introduction de la partie 5, nos tas fusionnables sont par défaut des tas min fusionnables auxquels s'appliquent donc les opérations MINIMUM, EXTRAIRE-MIN et DIMINUER-CLÉ. On pourrait aussi définir un *tas max fusionnable* doté des opérations MAXIMUM, EXTRAIRE-MAX et AUGMENTER-CLÉ.

s'avère indispensable, les tas binaires sont peu efficaces. La concaténation des deux tableaux contenant les tas binaires à fusionner, puis l'exécution de ENTASSER (voir exercice 6.2.2) font que UNION prend un temps $\Theta(n)$ dans le cas le plus défavorable.

Dans ce chapitre, on étudiera les « tas binomiaux », dont on peut aussi voir les bornes du temps d'exécution dans le cas le plus défavorable à la figure 19.1. En particulier, l'opération UNION ne requiert que $O(\lg n)$ pour fusionner deux tas binomiaux contenant au total n éléments.

Au chapitre 20, nous étudierons les tas de Fibonacci, qui bénéficient de bornes encore meilleures pour certaines opérations. Notez cependant que les temps d'exécution donnés pour les tas de Fibonacci à la figure 19.1 sont des bornes amorties et non des bornes dans le cas le plus défavorable pour chaque opération.

Procédure	Tas binaire (cas le plus défavorable)	Tas binomial (cas le plus défavorable)	Tas de Fibonacci (amorti)
CRÉER-TAS	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSÉRER	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRAIRE-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DIMINUER-CLÉ	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
SUPPRIMER	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Figure 19.1 Les temps d'exécution des opérations pour trois implémentations des tas fusionnables. Le nombre d'éléments du (des) tas au moment de l'opération est noté n .

Ce chapitre ignore les problèmes liés à l'allocation des nœuds avant une insertion et ceux liés à la libération des nœuds après une suppression. On suppose que ces détails sont pris en charge par le code qui appelle les procédures de tas.

Les tas binaires, les tas binomiaux et les tas de Fibonacci sont tous inefficaces pour l'opération RECHERCHER ; trouver un nœud contenant une clé donnée peut prendre beaucoup de temps. Pour cette raison, les opérations comme DIMINUER-CLÉ et SUPPRIMER qui font référence à un nœud particulier, ont besoin en entrée d'un pointeur sur le nœud en question. Comme on l'a vu dans l'étude des files de priorités (section 6.5), quand on emploie un tas fusionnable dans une application, on stocke souvent dans chaque élément du tas fusionnable une balise référant l'objet d'application associé, et l'on stocke dans chaque objet d'application une balise référant l'élément de tas fusionnable associé. La nature exacte de ces balises dépend de l'application et de son implémentation.

La section 19.1 définit les tas binomiaux après avoir défini les arbres binomiaux qui les composent. Elle introduit également une représentation particulière des tas binomiaux. La section 19.2 montre comment implémenter des opérations sur des tas binomiaux dans les bornes de temps données à la figure 19.1.

19.1 ARBRES BINOMIAUX ET TAS BINOMIAUX

Un tas binomial étant une collection d'arbres binomiaux, cette section commence par définir les arbres binomiaux et démontrer certaines propriétés fondamentales. Nous définirons ensuite les tas binomiaux et nous montrerons comment on peut les représenter.

19.1.1 Arbres binomiaux

Un *arbre binomial* B_k est un arbre ordonné (voir section B.5.2) défini récursivement. Comme on le voit à la figure 19.2(a), l'arbre binomial B_0 est constitué d'un nœud unique. L'arbre binomial B_k est constitué de deux arbres binomiaux B_{k-1} qui sont *reliés* entre eux : la racine de l'un est l'enfant le plus à gauche de la racine de l'autre. La figure 19.2(b) montre les arbres binomiaux B_0 à B_4 .

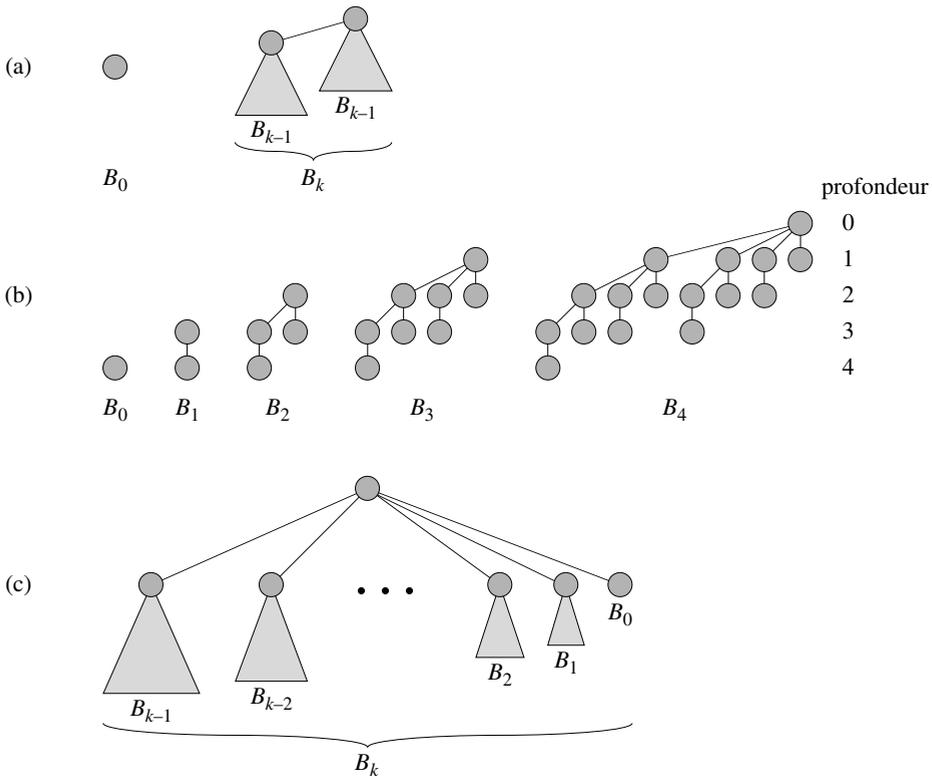


Figure 19.2 (a) Définition récursive de l'arbre binomial B_k . Les triangles représentent des sous-arbres enracinés. (b) Les arbres binomiaux B_0 à B_4 . La profondeur des nœuds de B_4 est représentée. (c) Une autre façon de voir l'arbre binomial B_k .

Le lemme suivant donne quelques propriétés des arbres binomiaux.

Lemme 19.1 (Propriétés des arbres binomiaux) *Pour un arbre binomial B_k ,*

- 1) *il existe 2^k nœuds,*
- 2) *la hauteur de l'arbre est k ,*
- 3) *il existe exactement $\binom{k}{i}$ nœuds à la profondeur i pour $i = 0, 1, \dots, k$,*
- 4) *la racine a le degré k , qui est supérieur à celui de tout autre nœud ; par ailleurs, si les enfants de la racine sont numérotés de la gauche vers la droite par $k - 1, k - 2, \dots, 0$, l'enfant i est la racine d'un sous-arbre B_i .*

Démonstration : La démonstration se fait par récurrence sur k . Pour chaque propriété, la base de la récurrence est l'arbre binomial B_0 . Vérifier que chaque propriété est valide pour B_0 est immédiat.

Pour l'étape inductive, on suppose que le lemme est valable pour B_{k-1} .

- 1) L'arbre binomial B_k est constitué de deux copies de B_{k-1} , et donc B_k contient $2^{k-1} + 2^{k-1} = 2^k$ nœuds.
- 2) A cause de la manière dont les deux instances de B_{k-1} sont reliées pour former B_k , la profondeur maximale d'un nœud dans B_k vaut un de plus que la profondeur maximale dans B_{k-1} . D'après l'hypothèse inductive, cette profondeur maximale est $(k - 1) + 1 = k$.
- 3) Soit $D(k, i)$ le nombre de nœuds à la profondeur i d'un arbre binomial B_k . Comme B_k est composé de deux instances de B_{k-1} reliées entre elles, un nœud de profondeur i dans B_{k-1} apparaît dans B_k une fois à la profondeur i et une fois à la profondeur $i + 1$. Autrement dit, le nombre de nœuds de profondeur i dans B_k est égal au nombre de nœuds de profondeur i dans B_{k-1} augmenté du nombre de nœuds à la profondeur $i - 1$ dans B_{k-1} . Donc,

$$\begin{aligned} D(k, i) &= D(k - 1, i) + D(k - 1, i - 1) \quad (\text{d'après l'hypothèse inductive}) \\ &= \binom{k - 1}{i} + \binom{k - 1}{i - 1} \quad (\text{d'après l'exercice C.1.7}) \\ &= \binom{k}{i}. \end{aligned}$$

- 4) Le seul nœud ayant un degré plus grand dans B_k que dans B_{k-1} est la racine, qui possède un enfant de plus que dans B_{k-1} . Comme la racine de B_{k-1} a pour degré $k - 1$, celle de B_k a pour degré k . Or, d'après l'hypothèse de récurrence, et comme le montre la figure 19.2(c), de gauche à droite, les enfants de la racine de B_{k-1} sont les racines de $B_{k-2}, B_{k-3}, \dots, B_0$. Lorsque B_{k-1} est relié à B_{k-1} , les enfants de la nouvelle racine sont donc les racines de $B_{k-1}, B_{k-2}, \dots, B_0$. □

Corollaire 19.2 *Le degré maximal d'un nœud quelconque dans un arbre binomial à n nœuds est $\lg n$.*

Démonstration : Immédiate d'après les propriétés 1 et 4 du lemme 19.1. □

Le terme « arbre binomial » vient de la propriété 3 du lemme 19.1, puisque les termes $\binom{k}{i}$ sont les coefficients binomiaux. Ce terme est mieux justifié par l'exercice 19.1.3.

19.1.2 Tas binomiaux

Un *tas binomial* T est un ensemble d'arbres binomiaux qui satisfait aux *propriétés des tas binomiaux*.

- 1) Chaque arbre binomial de T obéit à la *propriété de tas min* : la clé d'un nœud est supérieure ou égale à la clé de son parent. On dit d'un tel arbre qu'il est *trié en tas min*.
- 2) Quel que soit l'entier k positif, il existe dans T un arbre binomial au plus dont la racine a le degré k .

La première propriété nous dit que la racine d'un arbre trié en tas min contient la plus petite clé de l'arbre.

La deuxième propriété implique qu'un tas binomial T à n nœuds est constitué d'au plus $\lfloor \lg n \rfloor + 1$ arbres binomiaux. En effet, observez que la représentation binaire de n comporte $\lfloor \lg n \rfloor + 1$ bits, notés par exemple $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \dots, b_0 \rangle$, de sorte que $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. D'après la propriété 1 du lemme 19.1, l'arbre binomial B_i apparaît dans T si et seulement si le bit $b_i = 1$. Donc, le tas binomial T contient au plus $\lfloor \lg n \rfloor + 1$ arbres binomiaux.

La figure 19.3(a) montre un tas binomial T à 13 nœuds. La représentation binaire de 13 est $\langle 1101 \rangle$ et T est constitué des arbres binomiaux triés en tas min B_3 , B_2 et B_0 , ayant respectivement 8, 4 et 1 nœuds, pour un total de 13 nœuds.

a) Représentation des tas binomiaux

Comme le montre la figure 19.3(b), chaque arbre binomial d'un tas binomial est stocké dans la représentation « enfant de gauche, frère de droite » définie à la section 10.4. Chaque nœud comporte un champ *clé* et des données satellite spécifiques à l'application. En plus, chaque nœud x contient des pointeurs vers son parent ($p[x]$), vers son enfant le plus à gauche ($enfant[x]$) et vers le frère qui se trouve immédiatement à sa droite ($frère[x]$). Si le nœud x est une racine, $p[x] = \text{NIL}$. Si x n'a pas d'enfant, $enfant[x] = \text{NIL}$ et si x est l'enfant le plus à droite de son parent, $frère[x] = \text{NIL}$. Chaque nœud x contient également le champ *degré* $[x]$, qui représente le nombre d'enfants de x .

Toujours d'après la figure 19.3, les racines des arbres binomiaux d'un tas binomial sont organisées en une liste chaînée, appelée *liste des racines*. Les degrés des racines augmentent strictement lorsqu'on parcourt la liste des racines. D'après la deuxième propriété des tas binomiaux, dans un tas binomial à n nœuds, les degrés des racines sont un sous-ensemble de $\{0, 1, \dots, \lfloor \lg n \rfloor\}$. Le champ *frère* a une signification différente selon que le nœud est une racine ou non. Si x est une racine, $frère[x]$ pointe sur la racine suivante dans la liste des racines. (Comme d'habitude, $frère[x] = \text{NIL}$ si x est la dernière racine de la liste.)

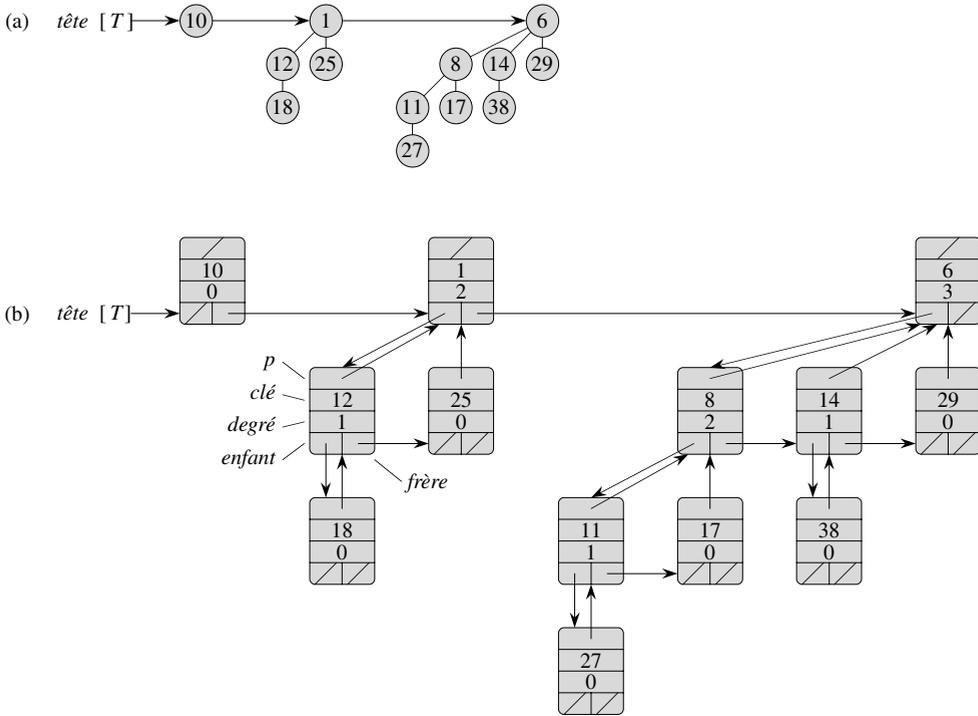


Figure 19.3 Un tas binomial T à $n = 13$ nœuds. (a) Le tas est constitué des arbres binomiaux B_0 , B_2 et B_3 , qui comportent respectivement 1, 4 et 8 nœuds, pour un total de $n = 13$ nœuds. Comme chaque arbre binomial est trié en tas min, la clé d'un nœud quelconque est supérieure ou égale à celle de son parent. La liste des racines est aussi représentée ; c'est une liste chaînée de racines par ordre de degrés croissant. (b) Une représentation plus détaillée du tas binomial T . Chaque arbre binomial est stocké dans la représentation « enfant de gauche, frère de droite » et chaque nœud contient son degré.

On peut accéder à un tas binomial T par le champ $tête[T]$, qui est un simple pointeur sur le premier élément de la liste des racines de T . Si le tas binomial T n'a pas d'éléments, $tête[T] = \text{NIL}$.

Exercices

19.1.1 On suppose que x est un nœud d'un arbre binomial d'un tas binomial et que $frère[x] \neq \text{NIL}$. Si x n'est pas une racine, quelle relation existe-t-il entre $degré[frère[x]]$ et $degré[x]$? Et si x est une racine ?

19.1.2 Si x est un nœud non racine d'un arbre binomial d'un tas binomial, quelle relation existe-t-il entre $degré[p[x]]$ et $degré[x]$?

19.1.3 On suppose qu'on étiquette les nœuds d'un arbre binomial B_k en binaire par un parcours postfixe, comme dans la figure 19.4. On considère un nœud x étiqueté l de profondeur i , et on pose $j = k - i$. Montrer que le nombre de 1 que contient x dans sa représentation binaire est égal à j . Combien existe-t-il de chaînes binaires d'ordre k contenant un nombre de 1 exactement égal à j ? Montrer que le degré de x est égal au nombre de 1 situés à la droite du 0 le plus à droite dans la représentation binaire de l .

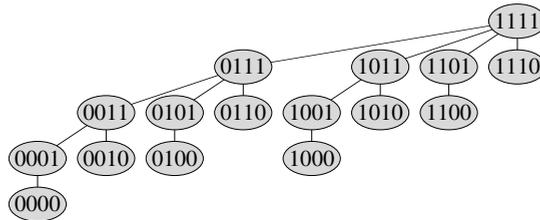


Figure 19.4 L'arbre binomial B_4 dont les nœuds sont étiquetés en binaire par un parcours postfixe.

19.2 OPÉRATIONS SUR LES TAS BINOMIAUX

Dans cette section, on montre comment effectuer les opérations sur les tas binomiaux dans les bornes de temps données à la figure 19.1. Nous n'établirons que les bornes supérieures. Les minorants seront laissés en exercice (voir exercice 19.2.10).

a) Création d'un tas binomial

Pour construire un tas binomial vide, la procédure `CRÉER-TAS-BINOMIAL` se contente d'allouer puis de retourner un objet T , où $tête[T] = \text{NIL}$. Son temps d'exécution est $\Theta(1)$.

b) Trouver la clé minimale

La procédure `MINIMUM-TAS-BINOMIAL` retourne un pointeur sur le nœud de clé minimale dans un tas binomial T à n nœuds. Cette implémentation suppose qu'aucune clé n'a la valeur ∞ . (Voir exercice 19.2.5.)

`MINIMUM-TAS-BINOMIAL(T)`

```

1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow t\grave{e}te[T]$ 
3   $min \leftarrow \infty$ 
4  tant que  $x \neq \text{NIL}$ 
5      faire si  $cl\grave{e}[x] < min$ 
6          alors  $min \leftarrow cl\grave{e}[x]$ 
7               $y \leftarrow x$ 
8           $x \leftarrow fr\grave{e}re[x]$ 
9  retourner  $y$ 
```

Puisqu'un tas binomial est trié en tas min, la clé minimale se trouve dans un nœud racine. La procédure `MINIMUM-TAS-BINOMIAL` teste toutes les racines, dont le nombre vaut au plus $\lfloor \lg n \rfloor + 1$, en plaçant le minimum courant dans *min* et un pointeur sur ce minimum courant dans *y*. Lorsqu'elle est appelée sur le tas binomial de la figure 19.3, `MINIMUM-TAS-BINOMIAL` retourne un pointeur sur le nœud de clé 1. Comme il y a au plus $\lfloor \lg n \rfloor + 1$ racines à tester, le temps d'exécution de `MINIMUM-TAS-BINOMIAL` est $O(\lg n)$.

c) Union de deux tas binomiaux

L'opération qui consiste à unir deux tas binomiaux sert de sous-programme pour la plupart des opérations restant à étudier. La procédure `UNION-TAS-BINOMIAUX` relie de façon répétée des arbres binomiaux dont les racines ont le même degré. La procédure suivante relie l'arbre B_{k-1} de racine *y* à l'arbre B_{k-1} de racine *z* ; autrement dit, il fait de *z* le parent de *y*. Le nœud *z* devient donc la racine d'un arbre B_k .

`LIEN-BINOMIAL(y, z)`

- 1 $p[y] \leftarrow z$
- 2 $frère[y] \leftarrow enfant[z]$
- 3 $enfant[z] \leftarrow y$
- 4 $degré[z] \leftarrow degré[z] + 1$

La procédure `LIEN-BINOMIAL` fait du nœud *y* la nouvelle tête de la liste chaînée des enfant du nœud *z*, et ce en un temps $O(1)$. Cela est rendu possible par la représentation « enfant de gauche, frère de droite » de chaque arbre binomial, qui correspond à la propriété hiérarchique de l'arbre ; dans un arbre B_k , l'enfant de la racine qui est le plus à gauche est la racine d'un arbre B_{k-1} .

La procédure suivante réunit les tas binomiaux T_1 et T_2 et retourne le tas résultant. Elle détruit, ce faisant, les représentations de T_1 et T_2 . En plus de `LIEN-BINOMIAL`, la procédure fait appel à une procédure auxiliaire, `FUSIONNER-TAS-BINOMIAUX`, qui fusionne les listes de racines de T_1 et T_2 pour en faire une seule liste chaînée, triée par ordre de degrés croissant. La procédure `FUSIONNER-TAS-BINOMIAUX`, dont le pseudo-code est laissé en exercice (voir exercice 19.2.1), ressemble à la procédure `FUSIONNER` de la section 2.3.1.

`FUSIONNER-TAS-BINOMIAUX(T_1, T_2)`

- 1 $T \leftarrow \text{CRÉER-TAS-BINOMIAL}()$
- 2 $tête[T] \leftarrow \text{FUSIONNER-TAS-BINOMIAUX}(T_1, T_2)$
- 3 libère objets T_1 et T_2 , mais pas les listes vers lesquelles ils pointent
- 4 **si** $tête[T] = \text{NIL}$
- 5 **alors retourner** T
- 6 $avant-x \leftarrow \text{NIL}$
- 7 $x \leftarrow tête[T]$
- 8 $après-x \leftarrow frère[x]$

```

9  tant que après-x ≠ NIL
10  faire si (degré[x] ≠ degré[après-x]) ou
      (frère[après-x] ≠ NIL et degré[frère[après-x]] = degré[x])
11      alors avant-x ← x                                     ▷ Cas 1 et 2
12      x ← après-x                                          ▷ Cas 1 et 2
13      sinon si clé[x] ≤ clé[après-x]
14          alors frère[x] ← frère[après-x]                 ▷ Cas 3
15          LIEN-BINOMIAL(après-x, x)                       ▷ Cas 3
16      sinon si avant-x = NIL                               ▷ Cas 4
17          alors tête[T] ← après-x                         ▷ Cas 4
18          sinon frère[avant-x] ← après-x                 ▷ Cas 4
19          LIEN-BINOMIAL(x, après-x)                       ▷ Cas 4
20          x ← après-x                                     ▷ Cas 4
21      après-x ← frère[x]
22  retourner T

```

La figure 19.5 montre un exemple de UNION-TAS-BINOMIAUX où les quatre cas prévus dans le pseudo-code se présentent.

La procédure UNION-TAS-BINOMIAUX se déroule en deux phases. La première, réalisée par l'appel à FUSIONNER-TAS-BINOMIAUX, fusionne les listes de racines des tas binomiaux T_1 et T_2 en une seule liste chaînée T , triée par ordre de degrés croissant. Cependant, pour chaque degré, il peut y avoir au plus deux racines ayant ce degré. La seconde phase relie donc les racines de degré égal, jusqu'à ce qu'il n'y ait plus qu'une seule racine au maximum par degré. Comme la liste chaînée T est triée par degré, les opérations de lien peuvent toutes s'effectuer rapidement.

Plus précisément, la procédure fonctionne de la manière suivante. Les lignes 1–3 commencent par fusionner les listes de racines des tas binomiaux T_1 et T_2 pour en faire une liste de racines unique T . Les listes de racines de T_1 et T_2 sont triées par ordre de degrés strictement croissant et FUSIONNER-TAS-BINOMIAUX retourne une liste de racines T qui est triée par degrés croissants. Si les listes de racines de T_1 et T_2 ont m racines à elles deux, FUSIONNER-TAS-BINOMIAUX s'exécute en $O(m)$ puisqu'elle examine à tour de rôle les racines qui sont en tête de chaque liste, concaténant la racine de plus bas degré à la liste des racines résultante puis supprimant cette racine de sa liste d'origine.

La procédure UNION-TAS-BINOMIAUX initialise ensuite quelques pointeurs vers la liste de racines de T . S'il s'avère que les deux tas binomiaux à réunir sont vides, elle se contente de rendre la main en (lignes 4–5). A partir de la ligne 6, on sait donc que T possède au moins une racine. Pendant la procédure, on gère trois pointeurs pointant vers la liste de racines :

- x pointe sur la racine en cours d'examen,
- $avant-x$ pointe sur la racine qui précède x dans la liste des racines : $frère[avant-x] = x$ (au début x n'a pas de prédécesseur, donc on part avec $avant-x$ réglé sur NIL), et

– après- x pointe sur la racine qui suit x dans la liste : $frère[x] = après-x$.

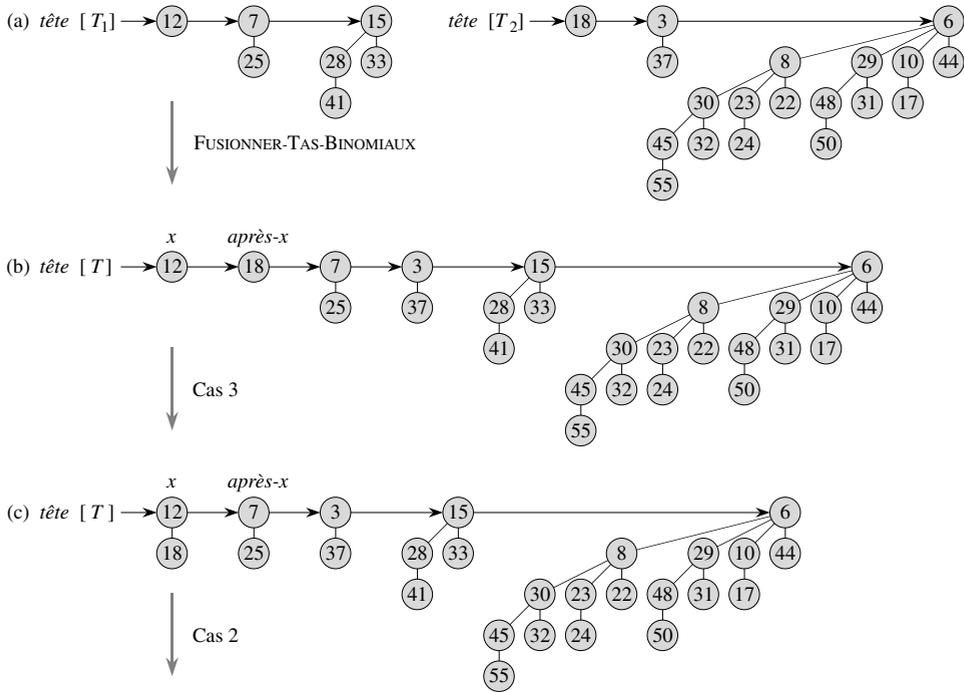
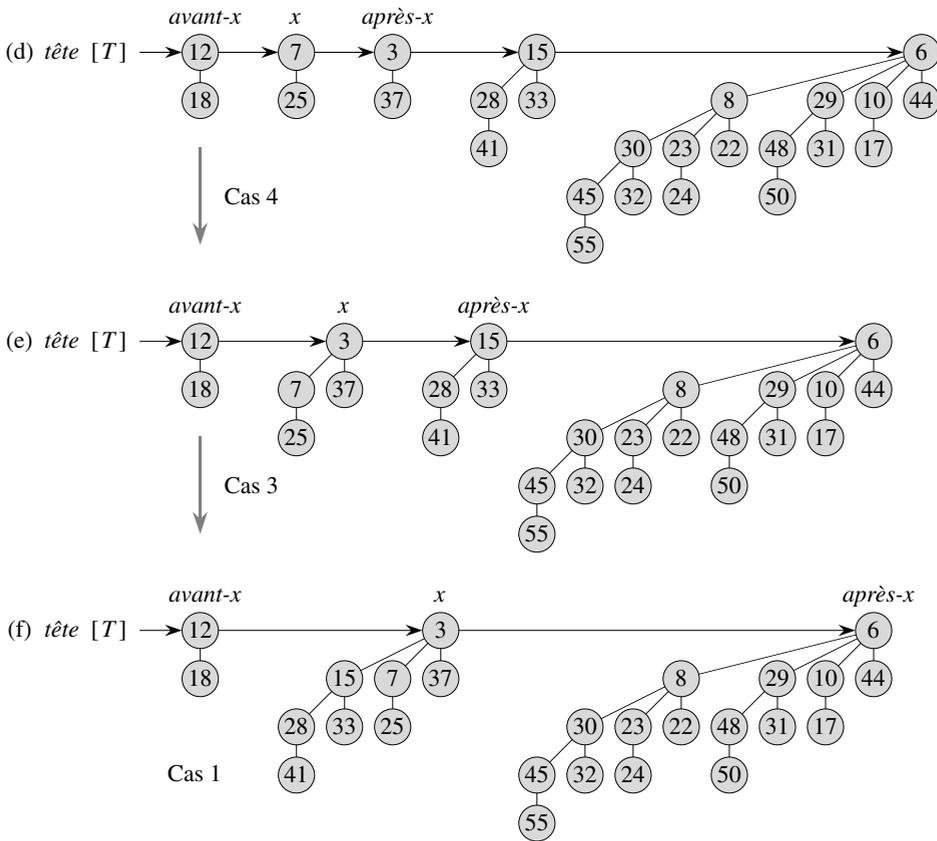


Figure 19.5 L'exécution de UNION-TAS-BINOMIAUX. (a) Les tas binomiaux T_1 et T_2 . (b) Le tas binomial T est le résultat de FUSIONNER-TAS-BINOMIAUX(T_1, T_2). Au départ, x est la première racine de la liste des racines de T . Comme x et $après-x$ ont pour degré 0 et $clé[x] < clé[après-x]$, le cas 3 s'applique. (c) Après le lien, x est la première de trois racines de même degré et le cas 2 s'applique. (d) Après que tous les pointeurs ont descendu d'une position dans la liste des racines, le cas 4 s'applique, puisque x est la première de deux racines de même degré. (e) Après le lien, le cas 3 s'applique. (f) Après un autre lien, le cas 1 s'applique, puisque x a pour degré 3 et $après-x$ a pour degré 4. Cette itération de la boucle **tant que** est la dernière, car après le déplacement des pointeurs d'une position vers le bas, $après-x = NIL$.

Au départ, pour un degré donné, la liste des racines de T contient au plus deux racines ayant ce degré : comme T_1 et T_2 étaient des tas binomiaux, ils avaient chacun une seule racine au plus ayant un degré donné. Par ailleurs, FUSIONNER-TAS-BINOMIAUX garantit que, si deux racines de T ont le même degré, elles sont adjacentes dans la liste des racines.

En fait, pendant l'exécution de UNION-TAS-BINOMIAUX, il pourrait exister en même temps trois racines ayant un degré donné dans la liste des racines de T . Nous verrons bientôt comment pourrait se présenter une telle situation. A chaque itération de la boucle **tant que** des lignes 9–21, la décision de relier x et $après-x$ se prend donc en fonction de leurs degrés respectifs et éventuellement du degré de $frère[après-x]$.



La boucle a pour invariant le fait qu’au début du corps de la boucle, x et $après-x$ sont différents de NIL. (Voir exercice 19.2.4 pour un invariant de boucle plus précis.)

Le cas 1, montré à la figure 19.6(a), se produit quand $degré[x] \neq degré[après-x]$, c’est-à-dire quand x est la racine d’un arbre B_k et $après-x$ est la racine d’un arbre B_l pour un certain $l > k$. Les lignes 11–12 gèrent ce cas. Comme x et $après-x$ ne sont pas reliés, on se contente de faire descendre les pointeurs d’une position dans la liste. La mise à jour de $après-x$ pour le faire pointer sur le nœud qui suit le nouveau nœud x est gérée à la ligne 21, qui est commune à tous les cas.

Le cas 2, montré à la figure 19.6(b), se produit quand x est la première de trois racines ayant le même degré, c’est-à-dire quand

$$degré[x] = degré[après-x] = degré[frère[après-x]] .$$

Ce cas est géré de la même façon que le cas 1 : on se contente de faire descendre les pointeurs d’une position dans la liste. L’itération suivante exécute le cas 3 ou le cas 4 pour combiner les seconde et troisième des trois racines de même degré. La ligne 10 vérifie si on est dans le cas 1 ou 2, et les lignes 11–12 traitent les deux cas.

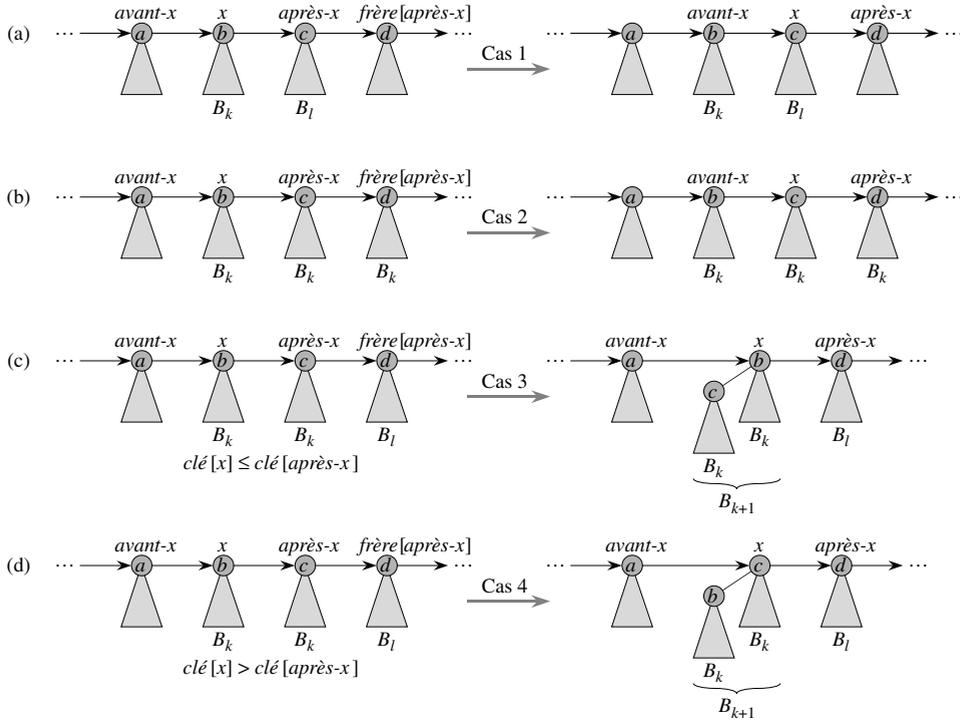


Figure 19.6 Les quatre cas de UNION-TAS-BINOMIAUX. Les étiquettes a , b , c et d ne servent qu'à identifier les racines mises en jeu ; elles n'indiquent pas les degrés, ni les clés de ces racines. Dans chaque cas, x est la racine d'un arbre B_k et $l > k$. (a) Cas 1 : $\text{degré}[x] \neq \text{degré}[\text{après-x}]$. Les pointeurs descendent d'une position dans la liste des racines. (b) Cas 2 : $\text{degré}[x] = \text{degré}[\text{après-x}] = \text{degré}[\text{frère}[\text{après-x}]]$. Là encore, les pointeurs descendent d'une position dans la liste et l'itération suivante exécute le cas 3 ou le cas 4. (c) Cas 3 : $\text{degré}[x] = \text{degré}[\text{après-x}] \neq \text{degré}[\text{frère}[\text{après-x}]]$ et $\text{clé}[x] \leq \text{clé}[\text{après-x}]$. On supprime après-x de la liste des racines et on le relie à x , pour créer un arbre B_{k+1} . (d) Cas 4 : $\text{degré}[x] = \text{degré}[\text{après-x}] \neq \text{degré}[\text{frère}[\text{après-x}]]$ et $\text{clé}[\text{après-x}] \leq \text{clé}[x]$. On retire x de la liste des racines et on le relie à après-x , pour créer là aussi un arbre B_{k+1} .

Les cas 3 et 4 se produisent quand x est la première de deux racines de même degré, c'est-à-dire quand

$$\text{degré}[x] = \text{degré}[\text{après-x}] \neq \text{degré}[\text{frère}[\text{après-x}]] .$$

Ces cas peuvent survenir dans n'importe quelle itération, mais l'un d'eux se produit systématiquement juste après le cas 2. Dans les cas 3 et 4, on relie x et après-x . Les deux cas se distinguent selon que c'est x ou après-x qui possède la plus petite clé, ce qui détermine celui des deux nœuds qui sera la racine une fois la liaison effectuée.

Dans le cas 3, montré à la figure 19.6(c), $\text{clé}[x] \leq \text{clé}[\text{après-x}]$ et donc après-x est relié à x . La ligne 14 supprime après-x de la liste des racines et la ligne 15 fait de après-x l'enfant de x le plus à gauche.

Dans le cas 4, montré à la figure 19.6(d), *après-x* possède la plus petite clé et x est donc relié à *après-x*. Les lignes 16–18 suppriment x de la liste des racines, ce qui peut se faire de deux façons différentes, selon que x est la première racine de la liste (ligne 17) ou non (ligne 18). Puis, la ligne 19 fait de x l'enfant le plus à gauche de *après-x* et la ligne 20 met x à jour pour la prochaine itération.

Juste après le cas 3 ou le cas 4, l'initialisation préparatoire à l'itération suivante de la boucle **tant que** est identique. Nous venons de relier deux arbres B_k pour former un arbre B_{k+1} , sur lequel on fait maintenant pointer x . Comme il existait déjà zéro, un ou deux autres arbres B_{k+1} dans la liste des racines produite par FUSIONNER-TAS-BINOMIAUX, x est maintenant le premier parmi un, deux ou trois arbres B_{k+1} dans la liste des racines. Si x est le seul, l'itération suivante nous ramène au cas 1 : $\text{degré}[x] \neq \text{degré}[\text{après-x}]$. Si x est le premier de deux, on se retrouve dans le cas 3 ou le cas 4 à l'itération suivante. Et c'est seulement si x est le premier parmi trois que l'itération suivante nous fait entrer dans le cas 2.

Le temps d'exécution de UNION-TAS-BINOMIAUX est $O(\lg n)$, où n est le nombre total de nœuds contenus dans les tas binomiaux T_1 et T_2 . On peut le voir de la manière suivante. Soit n_1 le nombre de nœuds de T_1 et n_2 le nombre de nœuds de T_2 , de sorte que $n = n_1 + n_2$. Alors, T_1 contient au plus $\lfloor \lg n_1 \rfloor + 1$ racines et T_2 contient au plus $\lfloor \lg n_2 \rfloor + 1$ racines ; donc T contient au plus $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2 \lfloor \lg n \rfloor + 2 = O(\lg n)$ racines immédiatement après l'appel à FUSIONNER-TAS-BINOMIAUX. Le temps d'exécution de FUSIONNER-TAS-BINOMIAUX est donc en $O(\lg n)$. Chaque itération de la boucle **tant que** requiert un temps en $O(1)$ et il existe au plus $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ itérations car chacune d'elle, soit fait descendre les pointeurs d'une position dans la liste des racines de T , soit supprime une racine de la liste des racines. Le temps total est donc $O(\lg n)$.

d) Insertion d'un nœud

La procédure ci-dessous insère un nœud x dans un tas binomial T , en supposant que le nœud x a déjà été alloué et que $\text{clé}[x]$ a déjà été initialisée.

```
TAS-BINOMIAL-INSÉRER( $T, x$ )
1   $T' \leftarrow \text{CRÉER-TAS-BINOMIAL}()$ 
2   $p[x] \leftarrow \text{NIL}$ 
3   $\text{enfant}[x] \leftarrow \text{NIL}$ 
4   $\text{frère}[x] \leftarrow \text{NIL}$ 
5   $\text{degré}[x] \leftarrow 0$ 
6   $\text{tête}[T'] \leftarrow x$ 
7   $T \leftarrow \text{UNION-TAS-BINOMIAUX}(T, T')$ 
```

Cette procédure se contente de créer en temps $O(1)$ un tas binomial T' à un nœud, puis de l'unir au tas binomial T à n nœuds en temps $O(\lg n)$. L'appel à UNION-TAS-BINOMIAUX se charge de libérer le tas binomial temporaire T' . (Une

implémentation directe, qui ne fait pas appel à UNION-TAS-BINOMIAUX, est donnée à l'exercice 19.2.8.)

e) *Extraction du nœud de clé minimale*

La procédure suivante extrait le nœud de clé minimale d'un tas binomial T et retourne un pointeur sur le nœud extrait.

```
TAS-BINOMIAL-EXTRAIRE-MIN( $T$ )
1  trouver la racine  $x$  de clé minimale dans la liste des racines de  $T$ ,
   et supprimer  $x$  de la liste
2   $T' \leftarrow$  CRÉER-TAS-BINOMIAL()
3  inverser l'ordre de la liste chaînée des enfant de  $x$ ,
   et faire pointer  $tête[T']$  sur la tête de la liste résultante
4   $T \leftarrow$  UNION-TAS-BINOMIAUX( $T, T'$ )
5  retourner  $x$ 
```

Le fonctionnement de cette procédure est illustré à la figure 19.7. Le tas binomial T en entrée est montré à la figure 19.7(a). La figure 19.7(b) donne la situation après la ligne 1 : la racine x de clé minimale a été ôtée de la liste des racines de T . Si x est la racine d'un arbre B_k , alors d'après la propriété 4 du lemme 19.1, les enfant de x , en allant de la gauche vers la droite, sont les racines d'arbres $B_{k-1}, B_{k-2}, \dots, B_0$. La figure 19.7(c) montre que, via une inversion de la liste des enfant de x à la ligne 3, on obtient un tas binomial T' qui contient tous les nœuds de l'arbre de x , sauf x lui-même. Comme l'arbre de x a été supprimé de T en ligne 1, le tas binomial montré à la figure 19.7(d) qui résulte de l'union de T et T' en ligne 4, contient tous les nœuds présents initialement dans T , excepté x . Enfin, la ligne 5 retourne x .

Puisque chacune des lignes 1–4 consomme un temps $O(\lg n)$ si T contient n nœuds, TAS-BINOMIAL-EXTRAIRE-MIN s'exécute en $O(\lg n)$.

f) *Diminution d'une clé*

La procédure suivante diminue la clé d'un nœud x dans un tas binomial T et lui donne une nouvelle valeur k . Elle signale une erreur si k est supérieure à la valeur de la clé courante de x .

```
TAS-BINOMIAL-DIMINUER-CLÉ( $T, x, k$ )
1  si  $k > clé[x]$ 
2  alors erreur « La nouvelle clé est plus grande que la clé courante »
3   $clé[x] \leftarrow k$ 
4   $y \leftarrow x$ 
5   $z \leftarrow p[y]$ 
```

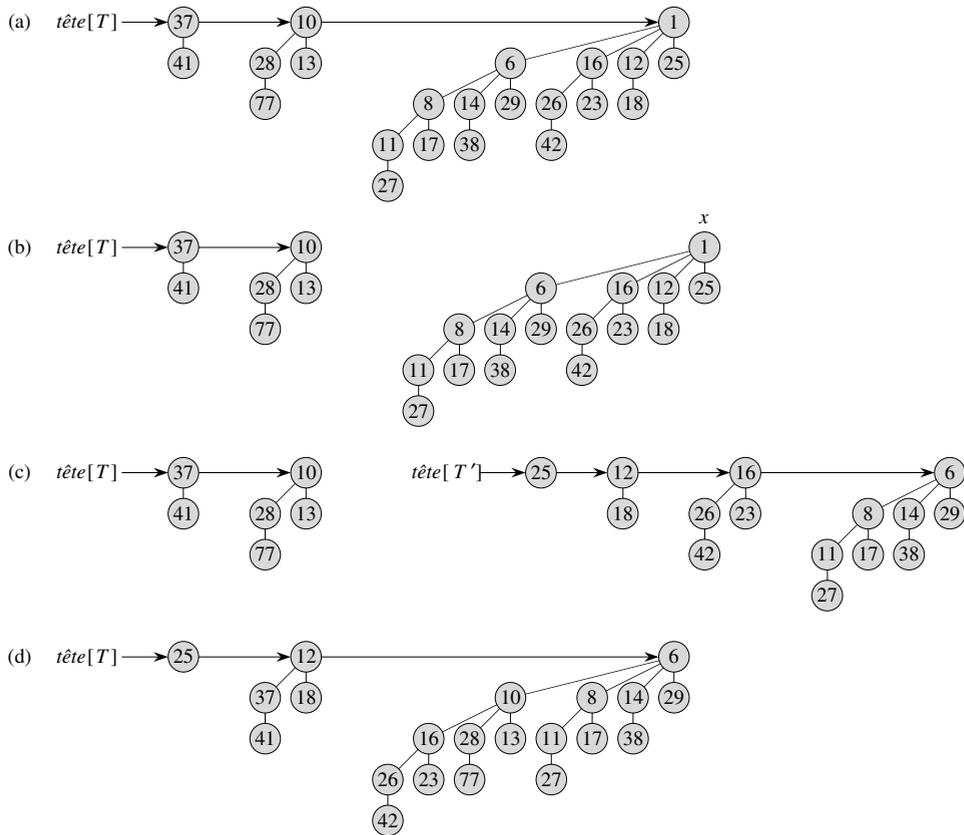


Figure 19.7 L'action de TAS-BINOMIAL-EXTRAIRE-MIN. (a) Un tas binomial T . (b) La racine x de clé minimale est supprimée de la liste des racines de T . (c) La liste chaînée des enfant de x est inversée, donnant ainsi un autre tas binomial T' . (d) Le résultat de l'union de T et T' .

```

6 tant que  $z \neq \text{NIL}$  et  $clé[y] < clé[z]$ 
7   faire permuter  $clé[y] \leftrightarrow clé[z]$ 
8     ▷ Si  $y$  et  $z$  ont des champs satellites, on les permute aussi.
9      $y \leftarrow z$ 
10     $z \leftarrow p[y]$ 

```

Comme on peut le voir à la figure 19.8, cette procédure diminue une clé de la même manière que dans un tas min binaire : en faisant remonter la clé dans le tas comme une bulle. Après avoir fait en sorte que la nouvelle clé ne soit pas plus grande que la clé courante, la procédure affecte la nouvelle clé à x , remonte l'arbre en commençant initialement par faire pointer y vers le nœud x . A chaque itération de la boucle **tant que** des lignes 6-10, $clé[y]$ est comparé à la clé de z , qui est le parent de y . Si y est la racine ou si $clé[y] \geq clé[z]$, l'arbre binomial a retrouvé son ordre de tas min. Sinon, le nœud y ne respecte pas l'ordre de tas min et sa clé est alors permutee avec

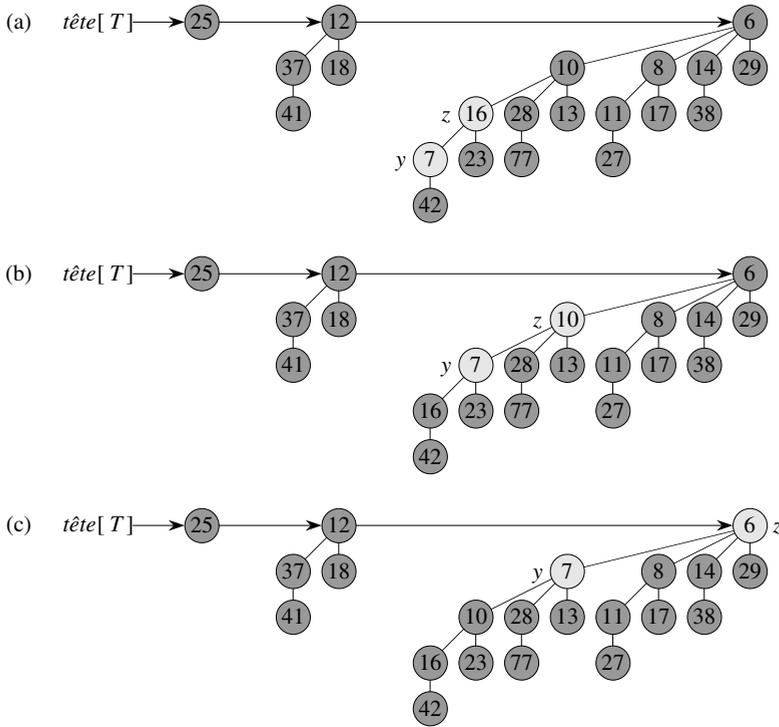


Figure 19.8 L'action de `TAS-BINOMIAL-DIMINUER-CLÉ`. (a) La situation juste avant la ligne?? de la première itération de la boucle **tant que**. Le nœud *y* a vu sa clé prendre la valeur 7 qui est inférieure à celle de la clé de *z*, parent de *y*. (b) Les clés des deux nœuds sont permutées et on peut voir l'état des lieux juste avant la ligne 6 de la deuxième itération. Les pointeurs *y* et *z* sont remontés d'un niveau dans l'arbre, mais l'ordre de tas min est encore violé. (c) Après une nouvelle permutation et une nouvelle remontée d'un niveau pour les pointeurs *y* et *z*, l'ordre de tas min est enfin respecté et la boucle **tant que** se termine.

celle de son parent, *z*, de même que les éventuelles informations satellites. La procédure initialise ensuite *y* à *z*, ce qui fait remonter d'un niveau dans l'arbre, et continue avec l'itération suivante.

La procédure `TAS-BINOMIAL-DIMINUER-CLÉ` s'exécute en $O(\lg n)$. D'après la propriété 2 du lemme 19.1, la profondeur maximale de *x* est $\lfloor \lg n \rfloor$, de sorte la boucle **tant que** des lignes 6-10 se répète au plus $\lfloor \lg n \rfloor$ fois.

g) Suppression d'une clé

Supprimer la clé et les informations satellites d'un nœud *x* contenu dans un tas binomial *T* se fait aisément en $O(\lg n)$. L'implémentation suivante suppose qu'aucun nœud du tas binomial ne comporte une clé de valeur $-\infty$.

```
TAS-BINOMIAL-SUPPRIMER(T, x)
1  TAS-BINOMIAL-DIMINUER-CLÉ(T, x,  $-\infty$ )
2  TAS-BINOMIAL-EXTRAIRE-MIN(T)
```

La procédure TAS-BINOMIAL-SUPPRIMER donne au nœud x la clé minimale unique dans tout le tas binomial, en lui affectant la valeur $-\infty$. (L'exercice 19.2.6 demande de gérer la situation où $-\infty$ ne peut pas être une valeur de clé, même temporairement.) Grâce à un appel à TAS-BINOMIAL-DIMINUER-CLÉ, cette clé est ensuite remontée jusqu'à la racine, ainsi que les informations satellite associées. Cette racine est alors supprimée de T par un appel à TAS-BINOMIAL-EXTRAIRE-MIN.

La procédure TAS-BINOMIAL-SUPPRIMER s'exécute en temps $O(\lg n)$.

Exercices

19.2.1 Écrire un pseudo code pour FUSIONNER-TAS-BINOMIAUX.

19.2.2 Montrer le tas binomial qui résulte de l'insertion d'un nœud de clé 24 dans le tas binomial de la figure 19.7(d).

19.2.3 Montrer le tas binomial qui résulte de la suppression du nœud de clé 28 du tas binomial de la figure 19.8(c).

19.2.4 Prouver la validité de UNION-TAS-BINOMIAUX en employant l'invariant de boucle que voici :

Au début de chaque itération de la boucle **tant que** des lignes 9–21, x pointe vers une racine qui est l'une des suivantes :

- la seule racine ayant le degré qu'elle a,
- la première des deux racines ayant le degré qu'elle a, ou
- la première ou la seconde des trois racines ayant le degré qu'elle a.

En outre, toutes les racines précédant le prédécesseur de x dans la liste des racines ont des degrés uniques au niveau de la liste des racines, et si le prédécesseur de x a un degré différent de celui de x , son degré est, lui aussi, unique dans la liste des racines. Enfin, les degrés des nœuds croissent de façon monotone quand on parcourt la liste des racines.

19.2.5 Expliquer pourquoi la procédure MINIMUM-TAS-BINOMIAL pourrait ne pas fonctionner correctement, si les clés pouvaient prendre la valeur ∞ . Réécrire le pseudo code pour que la procédure fonctionne correctement dans de telles situations.

19.2.6 On suppose qu'il est impossible de représenter la clé $-\infty$. Réécrire la procédure TAS-BINOMIAL-SUPPRIMER pour qu'elle gère cette situation. Son temps d'exécution devra rester $O(\lg n)$.

19.2.7 Étudier la relation entre l'insertion dans un tas binomial et l'incrémentement d'un compteur binaire. Étudier également les similitudes entre l'union de deux tas binomiaux et l'addition de deux nombres binaires.

19.2.8 A la lumière de l'exercice 19.2.7, réécrire TAS-BINOMIAL-INSÉRER pour insérer un nœud directement dans un tas binomial sans faire appel à UNION-TAS-BINOMIAUX.

19.2.9 Montrer que, si les listes de racines sont maintenues par ordre de degrés strictement décroissant (au lieu d'un ordre strictement croissant), chacune des opérations sur les tas binomiaux peuvent être implémentées sans modification du temps d'exécution asymptotique.

19.2.10 Trouver des entrées qui provoquent l'exécution des procédures TAS-BINOMIAL-EXTRAIRE-MIN, TAS-BINOMIAL-DIMINUER-CLÉ et TAS-BINOMIAL-SUPPRIMER en un temps $\Omega(\lg n)$. Expliquer pourquoi les temps d'exécution, dans le cas le plus défavorable, de TAS-BINOMIAL-INSÉRER, MINIMUM-TAS-BINOMIAL et UNION-TAS-BINOMIAUX sont $\Omega(\lg n)$ et non $\Omega(\lg n)$. (Voir problème 3.5.)

PROBLÈMES

19.1. Tas 2-3-4

Le chapitre 18 a introduit la notion d'arbre 2-3-4, où chaque nœud interne (autre qu'éventuellement la racine) possède deux, trois ou quatre enfant et où toutes les feuilles ont la même profondeur. Dans ce problème, nous allons implémenter des *tas 2-3-4*, capables de supporter les opérations des tas fusionnables.

Les différences entre les tas 2-3-4 et les arbres 2-3-4 sont les suivantes. Dans les tas 2-3-4, seules les feuilles comportent des clés et chaque feuille x contient exactement une clé, dans le champ $clé[x]$. Les clés ne sont pas particulièrement triées dans les feuilles ; autrement dit, de gauche à droite, les clés peuvent se trouver dans n'importe quel ordre. Chaque nœud interne x contient une valeur $petit[x]$ qui est égale à la plus petite clé présente dans un feuille quelconque du sous-arbre enraciné en x . La racine r contient un champ $hauteur[r]$ qui représente la hauteur de l'arbre. Enfin, les tas 2-3-4 sont prévus pour être conservés en mémoire principale, ce qui rend inutiles les opérations de lecture et d'écriture sur disque.

Implémenter les opérations suivantes pour les tas 2-3-4. Chacune des opérations des parties (a) à (e) doit s'exécuter en $O(\lg n)$ sur un tas 2-3-4 à n éléments. L'opération UNION de la partie (f) devra s'exécuter en $O(\lg n)$, où n est le nombre total d'éléments dans les deux tas d'entrée.

- a. MINIMUM, qui retourne un pointeur sur la feuille contenant la plus petite clé.
- b. DIMINUER-CLÉ, qui donne à la clé d'une feuille x donnée une valeur $k \leq clé[x]$.
- c. INSÉRER, qui insère une feuille x de clé k .
- d. SUPPRIMER, qui supprime une feuille x donnée.
- e. EXTRAIRE-MIN, qui extrait la feuille contenant la plus petite clé.
- f. UNION, qui réunit deux tas 2-3-4 et retourne un tas 2-3-4 unique, après avoir détruit les tas d'entrée.

19.2. Algorithme de l'arbre couvrant minimum et tas fusionnables

Le chapitre 23 présente deux algorithmes permettant de résoudre le problème de l'arbre couvrant de poids minimal dans un graphe non orienté. Ici, nous verrons comment on peut se servir de tas fusionnables pour concevoir un autre algorithme d'arbre couvrant minimum.

Soit $G = (S, A)$ un graphe non orienté connexe, avec une fonction de pondération $w : A \rightarrow \mathbf{R}$. On appelle $w(u, v)$ le poids de l'arête (u, v) . On souhaite trouver un arbre couvrant minimum pour G , c'est-à-dire un sous-ensemble acyclique $E \subseteq A$ qui relie tous les sommets de S et dont le poids total

$$w(E) = \sum_{(u,v) \in E} w(u, v)$$

est minimal.

Le pseudo code suivant, dont on peut prouver qu'il est correct à l'aide des techniques de la section 23.1, construit un arbre couvrant minimum E . Il gère une partition $\{S_i\}$ des sommets de S et, pour chaque ensemble S_i , un ensemble

$$A_i \subseteq \{(u, v) : u \in S_i \text{ ou } v \in S_i\}$$

d'arêtes incidentes pour les sommets de S_i .

ACM(G)

```

1   $E \leftarrow \emptyset$ 
2  pour chaque sommet  $v_i \in S[G]$ 
3      faire  $S_i \leftarrow \{v_i\}$ 
4           $A_i \leftarrow \{(v_i, v) \in A[G]\}$ 
5  tant que il y a plus d'un ensemble  $S_i$ 
6      faire choisir un ensemble quelconque  $S_i$ 
7          extraire l'arête de poids minimal  $(u, v)$  de  $A_i$ 
8          supposer, sans perte de généralité, que  $u \in S_i$  et  $v \in S_j$ 
9          si  $i \neq j$ 
10             alors  $E \leftarrow E \cup \{(u, v)\}$ 
11                  $S_i \leftarrow S_i \cup S_j$ , supprimant  $S_j$ 
12                  $A_i \leftarrow A_i \cup A_j$ 

```

Expliquer comment implémenter cet algorithme en utilisant des tas binomiaux pour gérer les ensembles de sommets et d'arêtes. Faut-il modifier la représentation d'un tas binomial ? Faut-il ajouter des opérations en plus des opérations de tas fusionnable données à la figure 19.1. Donner le temps d'exécution de votre implémentation.

NOTES

Les tas binomiaux ont été introduits en 1978 par Vuillemin [307]. Brown [49, 50] a étudié en détail leurs propriétés.

Chapitre 20

Tas de Fibonacci

Au chapitre 19, nous avons vu comment les tas binomiaux pouvaient supporter en temps $O(\lg n)$, dans le cas le plus défavorable, les opérations de tas fusionnable INSÉRER, MINIMUM, EXTRAIRE-MIN et UNION, plus les opérations DIMINUER-CLÉ et SUPPRIMER. Dans ce chapitre, nous étudierons les tas de Fibonacci, qui supportent les mêmes opérations mais avec l'avantage supplémentaire que les opérations qui ne nécessitent pas la suppression d'un élément s'exécutent avec un temps amorti $O(1)$.

D'un point de vue théorique, les tas de Fibonacci se révèlent particulièrement adaptés quand le nombre d'opérations EXTRAIRE-MIN et SUPPRIMER est petit par rapport aux autres opérations. Cette situation se produit dans de nombreuses applications. Par exemple, certains algorithmes de graphes sont susceptibles d'appeler DIMINUER-CLÉ une fois pour chaque arc. Pour les graphes denses, qui ont de nombreux arcs, le temps amorti $O(1)$ de chaque appel DIMINUER-CLÉ améliore grandement le temps $\Theta(\lg n)$ du cas le plus défavorable des tas binaires ou binomiaux. Certains algorithmes rapides pour des problèmes comme le calcul d'arbres couvrants minimaux (chapitre 23) et la recherche des plus courts chemins à origine unique (chapitre 24) s'appuient fondamentalement sur des tas de Fibonacci.

D'un point de vue pratique, toutefois, les facteurs constants et la programmation complexe des tas de Fibonacci les rendent moins intéressants que les tas binaires (ou k -aires) ordinaires pour la plupart des applications. Les tas de Fibonacci sont donc surtout intéressants d'un point de vue théorique. Si l'on inventait une structure de données beaucoup plus simple ayant les mêmes bornes temporelles amorties que les tas de Fibonacci, elle serait d'un grand intérêt pratique et non plus seulement théorique.

À l'instar d'un tas binomial, un tas de Fibonacci est une collection d'arbres. Les tas de Fibonacci sont, en fait, vaguement inspirés des tas binomiaux. Si les procédures DIMINUER-CLÉ et SUPPRIMER ne sont jamais invoquées dans un tas de Fibonacci, alors chaque arbre du tas ressemble à un arbre binomial. Les tas de Fibonacci ont une structure plus souple que les tas binomiaux, cependant, ce qui donne de meilleures bornes temporelles asymptotiques. Le travail de mise à jour de la structure peut être effectué en différé, si cela s'avère plus commode.

Comme les tables dynamiques de la section 17.4, les tas de Fibonacci offrent un bon exemple de structure de donnée conçue en ayant à l'esprit une analyse amortie. La compréhension intuitive et l'analyse des opérations de tas de Fibonacci que nous verrons dans ce chapitre se basent largement sur la méthode du potentiel vue à la section 17.3.

Ce chapitre suppose acquis les concepts du chapitre 19 sur les tas binomiaux. Les opérations utilisées ici sont spécifiées dans ce dernier chapitre, de même que le tableau de la figure 19.1, qui résume les bornes de temps d'exécution des opérations de tas binaire, de tas binomial et de tas de Fibonacci. Notre présentation de la structure de tas de Fibonacci s'appuie sur la présentation de la structure de tas binomial, et certaines des opérations effectuées sur les tas de Fibonacci sont semblables à celles faites sur les tas binomiaux.

Comme les tas binomiaux, les tas de Fibonacci ne sont pas prévus pour supporter efficacement l'opération RECHERCHER ; les opérations qui font référence à un nœud ont donc besoin d'avoir dans leurs paramètres d'entrée un pointeur vers ce nœud. Quand on utilise un tas de Fibonacci dans une application, on stocke souvent dans chaque élément du tas un handle référant l'objet d'application correspondant et l'on stocke dans chaque objet d'application un handle référant l'élément de tas de Fibonacci correspondant.

La section 20.1 définit les tas de Fibonacci, étudie leur représentation et présente la fonction potentiel utilisée pour leur analyse amortie. La section 20.2 montre comment implémenter les opérations de tas fusionnable et atteindre les bornes amorties données à la figure 19.1. Les deux dernières opérations, DIMINUER-CLÉ et SUPPRIMER, sont présentées à la section 20.3. Enfin, la section 20.4 termine en détaillant une partie fondamentale de l'analyse et explique l'origine étrange de la structure de données.

20.1 STRUCTURE DES TAS DE FIBONACCI

Comme les tas binomiaux, un *tas de Fibonacci* est un ensemble d'arbres ordonnés selon une structure de tas min. Toutefois, les arbres d'un tas de Fibonacci ne sont pas obligatoirement des arbres binomiaux. La figure 20.1(a) donne un exemple de tas de Fibonacci.

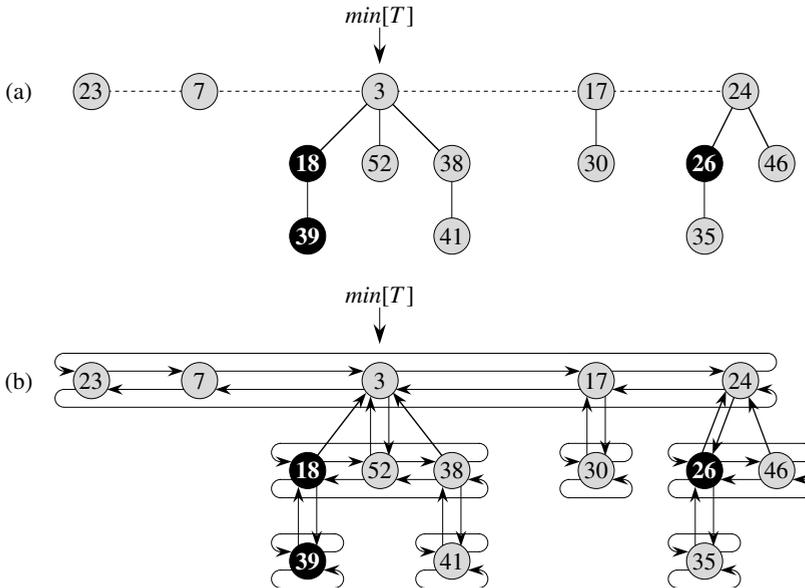


Figure 20.1 (a) Un tas de Fibonacci constitué de cinq arbres ordonnés en tas min et de 14 nœuds. La ligne pointillée représente la liste des racines. Le nœud minimal du tas est le nœud contenant la clé 3. Les trois nœuds marqués sont en noir. Le potentiel de ce tas de Fibonacci particulier est $5 + 2 \cdot 3 = 11$. (b) Une représentation plus complète, faisant apparaître les pointeurs p (flèches vers le haut), *enfant* (flèches vers le bas) et *gauche* et *droite* (flèches horizontales). Ces détails sont omis pour les autres figures de ce chapitre, puisque toutes les informations présentes ici peuvent être déterminées à partir de celles de la partie (a).

Contrairement aux arbres contenus dans les tas binomiaux, qui sont triés, ceux contenus dans les tas de Fibonacci sont enracinés mais non triés. Comme le montre la figure 20.1(b), chaque nœud x contient un pointeur $p[x]$ sur son parent et un pointeur *enfant* $[x]$ sur l'un quelconque de ses enfants. Les enfants de x sont reliés dans une liste doublement chaînée, que nous appellerons la *liste des enfants* de x . Chaque enfant y d'une liste de enfant possède les pointeurs *gauche* $[y]$ et *droite* $[y]$, qui pointent respectivement sur les frères de gauche et de droite de y . Si le nœud y est un enfant unique, alors *gauche* $[y] = droite[y] = y$. L'ordre dans lequel apparaissent les frères dans la liste des enfants est arbitraire.

Les listes circulaires doublement chaînées (voir section 10.2) ont deux avantages qui serviront pour le traitement des tas de Fibonacci. Primo, un nœud peut être supprimé en temps $O(1)$. Ensuite, étant données deux listes de ce type, on peut les concaténer (les « greffer » l'une sur l'autre) $O(1)$ pour obtenir une seule liste circulaire doublement chaînée en temps $O(1)$. Lorsque nous décrirons les opérations de tas de Fibonacci, nous nous y référerons de manière informelle, en laissant au lecteur les détails de leur implémentation.

Chaque nœud a deux autres champs qui nous seront utiles. Le nombre d'enfants contenus dans la liste des enfant d'un nœud x est conservé dans *degré* $[x]$. Le champ à

valeur booléenne *marqué*[x] indique si le nœud x a perdu un enfant depuis la dernière fois où x est devenu enfant d'un autre nœud. Les nœuds nouvellement créés ne sont pas marqués et un nœud x est « démarqué » chaque fois qu'il devient enfant d'un autre nœud. Jusqu'à ce que nous étudions l'opération DIMINUER-CLÉ à la section 20.3, nous nous contenterons d'initialiser tous les champs *marqué* à FAUX.

On peut accéder à un tas de Fibonacci T par un pointeur $\text{min}[T]$ pointant vers la racine d'un arbre contenant une clé minimale ; ce nœud est appelé le **nœud minimal** du tas de Fibonacci. Si un tas de Fibonacci T est vide, $\text{min}[T] = \text{NIL}$.

Les racines de tous les arbres d'un tas de Fibonacci sont reliées entre elles, à l'aide de leurs pointeurs *gauche* et *droite*, dans une liste doublement chaînée circulaire, appelée **liste des racines** du tas de Fibonacci. Le pointeur $\text{min}[T]$ pointe donc sur un nœud de la liste des racines dont la clé est minimale. L'ordre des arbres dans une liste des racines est arbitraire.

Nous nous appuyerons sur un autre attribut d'un tas de Fibonacci : le nombre total de nœuds présents dans T est conservé dans $n[T]$.

a) Fonction potentiel

Comme nous l'avons dit, nous utiliserons la méthode du potentiel de la section 17.3 pour analyser les performances des opérations de tas de Fibonacci. Pour un tas de Fibonacci T , on représente par $a(T)$ le nombre d'arbres contenus dans la liste des racines de T et par $m(T)$ le nombre de nœuds marqués dans T . Le potentiel du tas de Fibonacci T est alors défini par

$$\Phi(T) = a(T) + 2m(T) . \quad (20.1)$$

Par exemple, le potentiel du tas de Fibonacci montré à la figure 20.1 est $5 + 2 \cdot 3 = 11$. Le potentiel d'un ensemble de tas de Fibonacci est la somme des potentiels des tas de Fibonacci qui le constituent. On supposera qu'une unité de potentiel permet de payer une quantité de travail constante, la constante étant suffisamment grande pour couvrir le coût de n'importe quel travail partiel spécifique à temps constant que nous pourrions avoir à effectuer.

On suppose qu'une application utilisant les tas de Fibonacci commence sans tas. Le potentiel initial vaut donc 0 et, d'après l'équation (20.1), le potentiel reste ensuite toujours positif ou nul. D'après l'équation (17.3), un majorant du coût total amorti est donc un majorant du coût total réel de la séquence d'opérations.

b) Degré maximal

Les analyses amorties que nous effectuerons dans les sections suivantes de ce chapitre supposent qu'il existe un majorant connu $D(n)$ pour le degré maximal d'un nœud quelconque d'un tas de Fibonacci à n nœuds. L'exercice 20.2.3 montrera que, quand les opérations de tas fusionnables sont les seules à être supportées, alors $D(n) \leq \lceil \lg n \rceil$. A la section 20.3, nous montrerons que, quand DIMINUER-CLÉ et SUPPRIMER sont également supportées, alors $D(n) = O(\lg n)$.

20.2 OPÉRATIONS SUR LES TAS FUSIONNABLES

Dans cette section, on décrit et analyse les opérations sur les tas fusionnables basés sur des tas de Fibonacci. Si les opérations CRÉER-TAS, INSÉRER, MINIMUM, EXTRAIRE-MIN et UNION sont les seules à prendre en compte, alors chaque tas de Fibonacci est tout simplement une collection d'arbres binomiaux « non triés ». Un *arbre binomial non trié* ressemble à un arbre binomial et il est, lui aussi, défini récursivement. L'arbre binomial non trié U_0 est constitué d'un nœud unique et un arbre binomial non trié U_k est constitué de deux arbres binomiaux non triés U_{k-1} pour lequel la racine de l'un devient un enfant *quelconque* de la racine de l'autre. Le lemme 19.1, qui donne les propriétés des arbres binomiaux, est également valable pour les arbres binomiaux non triés, mais avec une légère variation dans la propriété 4 (voir exercice 20.2.2) :

4'. Pour l'arbre binomial non trié U_k , la racine a le degré k , qui est plus grand que celui de n'importe quel autre nœud. Les enfants de la racine sont les racines des sous-arbres U_0, U_1, \dots, U_{k-1} dans un certain ordre.

Donc, si un tas de Fibonacci à n nœuds est une collection d'arbres binomiaux non triés, alors $D(n) = \lg n$.

Pour les opérations de tas fusionnable sur les tas de Fibonacci, le principe est de différer le plus longtemps possible le travail à effectuer. On doit faire un compromis concernant les performances des implémentations des différentes opérations. Si le nombre d'arbres d'un tas de Fibonacci est petit, on peut rapidement déterminer lequel des nœuds restants doit devenir le nouveau nœud minimal lors d'une opération EXTRAIRE-MIN. Cependant, comme nous l'avons vu à l'exercice 19.2.10 pour les tas binomiaux, garantir que le nombre d'arbres est petit n'est pas gratuit ; l'insertion d'un nœud dans un tas binomial ou l'union de deux tas binomiaux peut prendre jusqu'à $\Omega(\lg n)$. Comme nous le verrons, on ne tente pas de consolider les arbres contenus dans un tas de Fibonacci lorsqu'on insère un nouveau nœud ou qu'on réunit deux tas. On réserve la consolidation pour l'opération EXTRAIRE-MIN, c'est-à-dire le moment où l'on a réellement besoin de trouver le nouveau nœud minimal.

c) Création d'un nouveau tas de Fibonacci

Pour créer un tas de Fibonacci vide, la procédure CRÉER-TAS-FIB alloue et retourne l'objet T , où $n[T] = 0$ et $\min[T] = \text{NIL}$; T ne contient aucun arbre. Comme $a(T) = 0$ et $m(T) = 0$, le potentiel du tas de Fibonacci vide est $\Phi(T) = 0$. Le coût amorti de CRÉER-TAS-FIB est donc égal à son coût réel $O(1)$.

d) Insertion d'un nœud

La procédure suivante insère un nœud x dans un tas de Fibonacci T , en supposant que le nœud a déjà été alloué et que $\text{clé}[x]$ n'est pas vide.

INSÉRER-TAS-FIB(T, x)

- 1 $degré[x] \leftarrow 0$
- 2 $p[x] \leftarrow \text{NIL}$
- 3 $enfant[x] \leftarrow \text{NIL}$
- 4 $gauche[x] \leftarrow x$
- 5 $droite[x] \leftarrow x$
- 6 $marqué[x] \leftarrow \text{FAUX}$
- 7 concaténer liste de racines contenant x et liste de racines T
- 8 **si** $min[T] = \text{NIL}$ ou $clé[x] < clé[min[T]]$
- 9 **alors** $min[T] \leftarrow x$
- 10 $n[T] \leftarrow n[T] + 1$

Après initialisation par les lignes 1–6 des champs du nœud x , créant ainsi sa propre liste circulaire doublement chaînée, la ligne 7 ajoute x à la liste des racines de T dans un temps $O(1)$. Ainsi, le nœud x devient un arbre ordonné de type tas min contenant un seul nœud, et donc un arbre binomial non ordonné, du tas de Fibonacci. Il n'a aucun enfant et n'est pas marqué. Les lignes 8–9 mettent ensuite à jour le pointeur vers le nœud minimal du tas de Fibonacci T , si nécessaire. Enfin, la ligne 10 incrémente $n[T]$ pour prendre en compte l'addition du nouveau nœud. La figure 20.2 montre l'insertion d'un nœud de clé 21 dans le tas de Fibonacci de la figure 20.1.

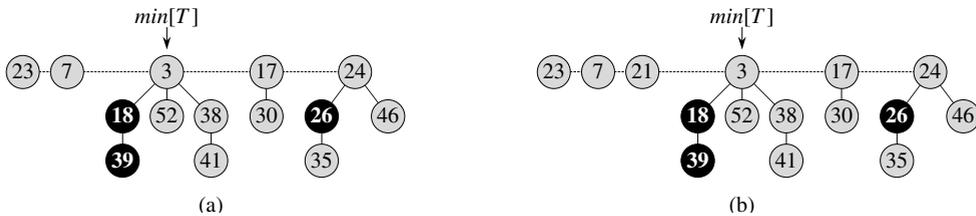


Figure 20.2 Insertion d'un nœud dans un tas de Fibonacci. (a) Un tas de Fibonacci T . (b) Le tas T après insertion du nœud de clé 21. Le nœud devient son propre arbre ordonné de type tas min et est ensuite ajouté à la liste des racines, ce qui en fait le frère gauche de la racine.

Contrairement à la procédure INSÉRER-TAS-BINOMIAL, INSÉRER-TAS-FIB n'essaye pas de consolider les arbres à l'intérieur du tas de Fibonacci. S'il y a k opérations INSÉRER-TAS-FIB consécutives, alors k arbres à un nœud sont ajoutés à la liste des racines.

Pour déterminer le coût amorti de INSÉRER-TAS-FIB, soit T le tas de Fibonacci en entrée et T' le tas de Fibonacci résultant. On a $a(T') = a(T) + 1$ et $m(T') = m(T)$, et l'augmentation de potentiel est

$$((a(T) + 1) + 2m(T)) - (a(T) + 2m(T)) = 1.$$

Comme le coût réel est $O(1)$, le coût amorti est $O(1) + 1 = O(1)$.

e) Recherche du nœud minimal

Le nœud minimal dans un tas de Fibonacci T est donné par le pointeur $\text{min}[T]$, ce qui permet de le retrouver dans un temps $O(1)$. Comme le potentiel de T n'est pas modifié, le coût amorti de cette opération est égal à son coût réel $O(1)$.

f) Union de deux tas de Fibonacci

La procédure suivante réunit les tas de Fibonacci T_1 et T_2 , détruisant T_1 et T_2 ce faisant. Elle se contente de concaténer les listes de racines de T_1 et T_2 , puis elle détermine le nouveau nœud minimal.

```

UNION-TAS-FIB( $T_1, T_2$ )
1   $T \leftarrow \text{CRÉER-TAS-FIB}()$ 
2   $\text{min}[T] \leftarrow \text{min}[T_1]$ 
3  concaténer liste de racines de  $T_2$  à celle de  $T$ 
4  si ( $\text{min}[T_1] = \text{NIL}$ ) ou ( $\text{min}[T_2] \neq \text{NIL}$  et  $\text{clé}[\text{min}[T_2]] < \text{clé}[\text{min}[T_1]]$ )
5    alors  $\text{min}[T] \leftarrow \text{min}[T_2]$ 
6   $n[T] \leftarrow n[T_1] + n[T_2]$ 
7  libérer objets  $T_1$  et  $T_2$ 
8  retourner  $T$ 

```

Les lignes 1–3 concatènent les listes de racines de T_1 et T_2 en une nouvelle liste T . Les lignes 2, 4 et 5 initialisent le nœud minimal de T et la ligne 6 affecte à $n[T]$ le nombre total de nœuds. Les objets T_1 et T_2 sont libérés à la ligne 7, et la ligne 8 retourne le tas de Fibonacci T final. Comme pour la procédure INSÉRER-TAS-FIB aucune consolidation des arbres n'intervient.

Le changement de potentiel est

$$\begin{aligned}
 \Phi(T) - (\Phi(T_1) + \Phi(T_2)) &= (a(T) + 2m(T)) - ((a(T_1) + 2m(T_1)) + (a(T_2) + 2m(T_2))) \\
 &= 0,
 \end{aligned}$$

car $a(T) = a(T_1) + a(T_2)$ et $m(T) = m(T_1) + m(T_2)$. Le coût amorti de UNION-TAS-FIB est donc égal à son coût réel $O(1)$.

g) Extraction du nœud minimal

Le processus d'extraction du nœud minimal est l'opération la plus compliquée parmi celles qui sont présentées dans cette section. C'est également à cet endroit que survient le travail de consolidation différée des arbres figurant dans la liste de racines. Le pseudo code suivant extrait le nœud minimal. On suppose par commodité que, lorsqu'un nœud est supprimé d'une liste chaînée, les pointeurs restant dans la liste sont mis à jour mais que ceux du nœud extrait restent inchangés. On fait aussi appel à la procédure auxiliaire CONSOLIDER, qui sera présentée bientôt.

```

EXTRAIRE-MIN-TAS-FIB( $T$ )
1   $z \leftarrow \text{min}[T]$ 
2  si  $z \neq \text{NIL}$ 
3      alors pour chaque enfant  $x$  de  $z$ 
4          faire ajouter  $x$  à la liste de racines de  $T$ 
5               $p[x] \leftarrow \text{NIL}$ 
6          supprimer  $z$  de la liste de racines de  $T$ 
7      si  $z = \text{droite}[z]$ 
8          alors  $\text{min}[T] \leftarrow \text{NIL}$ 
9          sinon  $\text{min}[T] \leftarrow \text{droite}[z]$ 
10         CONSOLIDER( $T$ )
11          $n[T] \leftarrow n[T] - 1$ 
12 retourner  $z$ 

```

Comme le montre la figure 20.3, EXTRAIRE-MIN-TAS-FIB commence par transformer en racine chaque enfant du nœud minimal et par supprimer le nœud minimal de la liste de racines. Puis, elle consolide la liste des racines en reliant entre elles les racines de degré égal, jusqu'à ce qu'il n'existe plus qu'une racine, au plus, pour chaque degré.

On commence en ligne 1 par sauvegarder un pointeur z sur le nœud minimal ; ce pointeur est retourné à la fin de la procédure. Si $z = \text{NIL}$, alors le tas de Fibonacci T est déjà vide et c'est terminé. Sinon, comme pour la procédure EXTRAIRE-MIN-TAS-BINOMIAL, on supprime le nœud z de T en transformant en racines tous les enfant de z aux lignes 3–5 (ils sont placés dans la liste des racines) et en supprimant z de la liste des racines à la ligne 6. Si $z = \text{droite}[z]$ après la ligne 6, alors z était le seul nœud de la liste de racines et il n'avait pas de enfant ; il ne reste donc plus qu'à vider la tête du tas de Fibonacci à la ligne 8, avant de retourner z . Dans le cas contraire, on fait en sorte que le pointeur $\text{min}[T]$ pointe vers un nœud de la liste de racines autre que z (dans ce cas $\text{droite}[z]$), qui ne va pas forcément être le nouveau nœud minimal quand EXTRAIRE-MIN-TAS-FIB aura fini. La figure 20.3(b) montre le tas de Fibonacci de la figure 20.3(a) après exécution de la ligne 9.

L'étape suivante, pendant laquelle on réduit le nombre d'arbres du tas de Fibonacci, est la **consolidation** de la liste des racines de T ; cela est effectué par l'appel CONSOLIDER(T). Consolider la liste des racines revient à exécuter de manière répétée les étapes suivantes, jusqu'à ce que toutes les racines de la liste aient une valeur *degré* distincte.

- 1) Trouver dans la liste des racines deux racines x et y ayant le même degré et pour lesquelles $\text{clé}[x] \leq \text{clé}[y]$.
- 2) **Relier** y à x : supprimer y de la liste des racines et faire de y un enfant de x . Cette opération est prise en charge par la procédure RELIER-TAS-FIB. Le champ $\text{degré}[x]$ est incrémenté et la marque éventuellement présente sur y est effacée.

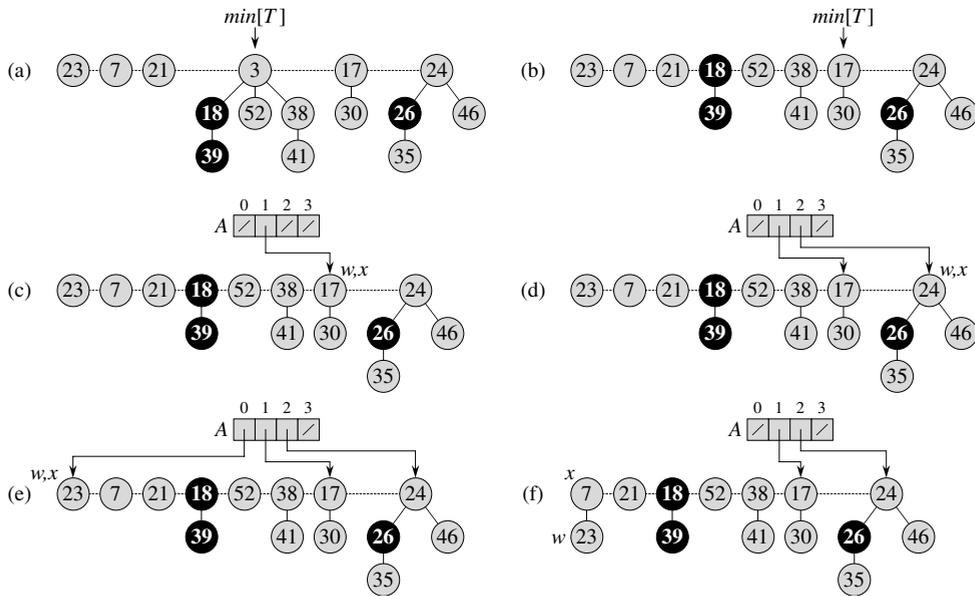
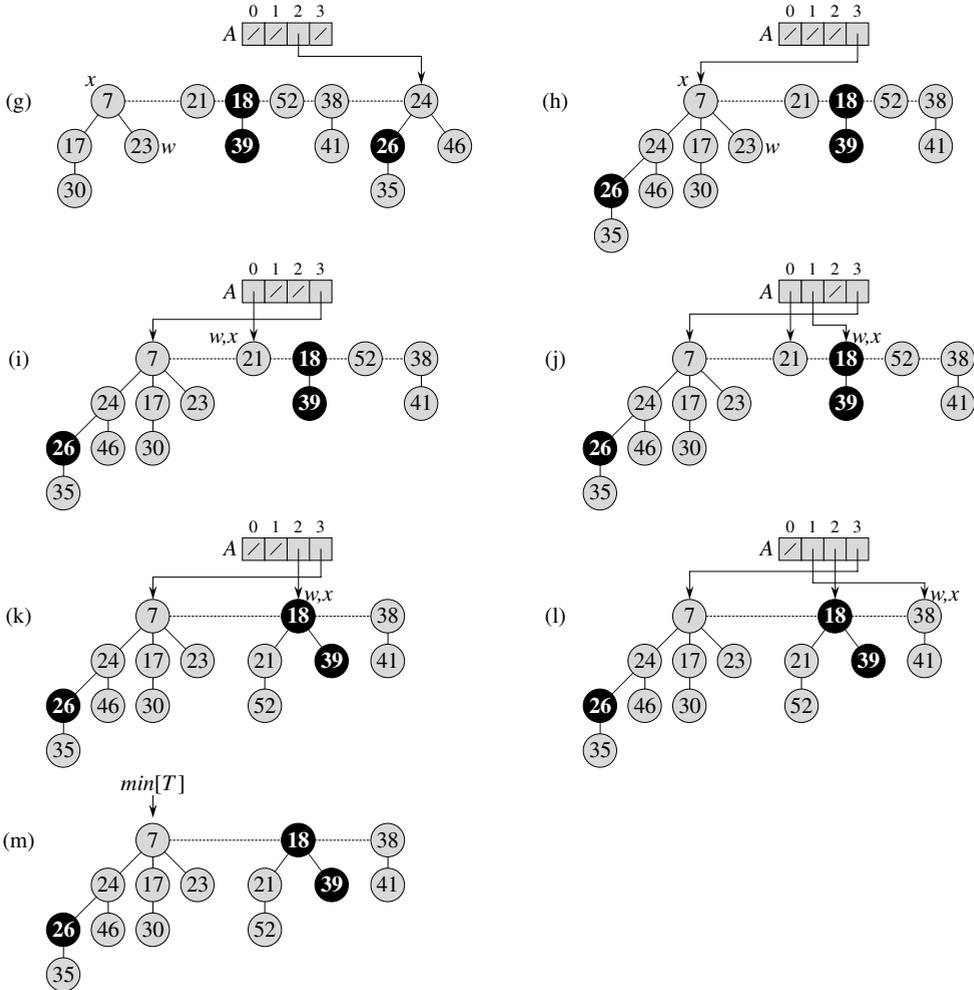


Figure 20.3 L'action de EXTRAIRE-MIN-TAS-FIB. (a) Un tas de Fibonacci T . (b) La situation après suppression du nœud minimal z de la liste des racines et après insertion de ses enfants dans la liste de racines. (c)–(e) Le tableau A et les arbres après chacune des trois premières itérations de la boucle **pour** des lignes 3–13 de la procédure CONSOLIDER. Le traitement de la liste des racines commence par le nœud pointé par $min[T]$ et suit les pointeurs *droite*. Chaque partie montre les valeurs de w et x à la fin d'une itération. (f)–(h) L'itération suivante de la boucle **pour**, avec les valeurs de w et x montrées à la fin de chaque itération de la boucle **tant que** des lignes 6–12. La partie (f) montre la situation après la première entrée dans la boucle **tant que**. Le nœud de clé 23 a été relié au nœud de clé 7, qui est maintenant pointé par x . Dans la partie (g), le nœud de clé 17 a été relié à celui de clé 7, qui est toujours pointé par x . Dans la partie (h), le nœud de clé 24 a été relié au nœud de clé 7. Comme aucun nœud n'était pointé précédemment par $A[3]$, à la fin de l'itération de la boucle **pour**, on fait pointer $A[3]$ sur la racine de l'arbre résultant. (i)–(l) La situation après chacune des quatre itérations suivantes de la boucle **pour**. (m) Le tas de Fibonacci T après reconstruction de la liste de racines à partir du tableau A et détermination du nouveau pointeur $min[T]$.



La procédure CONSOLIDER utilise un tableau auxiliaire $A[0 \dots D(n[T])]$; si $A[i] = y$, alors y est actuellement une racine pour laquelle $degré[y] = i$.

CONSOLIDER(T)

```

1  pour  $i \leftarrow 0$  à  $D(n[T])$ 
2    faire  $A[i] \leftarrow \text{NIL}$ 
3  pour chaque nœud  $w$  de la liste des racines de  $T$ 
4    faire  $x \leftarrow w$ 
5       $d \leftarrow \text{degré}[x]$ 
6    tant que  $A[d] \neq \text{NIL}$ 
7      faire  $y \leftarrow A[d]$ 
8        si  $\text{clé}[x] > \text{clé}[y]$ 
9          alors permuter  $x \leftrightarrow y$ 
10       RELIER-TAS-FIB( $T, y, x$ )
11        $A[d] \leftarrow \text{NIL}$ 
12        $d \leftarrow d + 1$ 
13      $A[d] \leftarrow x$ 
14   $\text{min}[T] \leftarrow \text{NIL}$ 
15  pour  $i \leftarrow 0$  à  $D(n[T])$ 
16    faire si  $A[i] \neq \text{NIL}$ 
17      alors ajouter  $A[i]$  à la liste des racines de  $T$ 
18      si  $\text{min}[T] = \text{NIL}$  ou  $\text{clé}[A[i]] < \text{clé}[\text{min}[T]]$ 
19        alors  $\text{min}[T] \leftarrow A[i]$ 

```

RELIER-TAS-FIB(T, y, x)

```

1  supprimer  $y$  de la liste des racines de  $T$ 
2  faire de  $y$  un enfant de  $x$ , incrémenter  $\text{degré}[x]$ 
3   $\text{marqué}[y] \leftarrow \text{FAUX}$ 

```

Dans le détail, la procédure CONSOLIDER fonctionne de la manière suivante. Les lignes 1–2 initialisent A en affectant à chaque élément NIL. La boucle **pour** des lignes 3–13 traite chaque racine w de la liste de racines. Après traitement de chaque racine w , celle-ci se retrouve dans un arbre enraciné en un certain nœud x , qui peut ou non être identique à w . Parmi les racines traitées, aucune autre n’aura le même degré que x , et donc on fait pointer l’élément $A[\text{degré}[x]]$ vers x . Quand cette boucle **pour** se terminera, il restera au plus une racine pour chaque degré, et le tableau A pointera alors vers chaque racine restante.

La boucle **tant que** des lignes 6–12 relie, de manière répétée, la racine x de l’arbre contenant le nœud w à un autre arbre dont la racine a le même degré que x , et ce jusqu’à ce qu’il n’y ait plus aucune racine qui ait le même degré. Cette boucle **tant que** conserve l’invariant que voici : Au début de chaque itération de la boucle **tant que**, $d = \text{degré}[x]$.

Nous utiliserons ainsi cet invariant :

Initialisation : La ligne 5 assure que l’invariant est vrai la première fois que l’on entre dans la boucle.

Conservation : À chaque itération de la boucle **tant que**, $A[d]$ pointe vers une certaine racine y . Comme $d = \text{degré}[x] = \text{degré}[y]$, on doit relier x et y . Celui de x et y qui a la plus petite clé devient le parent de l'autre dans le cadre de l'opération de liaison ; donc, les lignes 8–9 permutent, si nécessaire, les pointeurs pointant vers x et y . Ensuite, on relie y à x via l'appel RELIER-TAS-FIB(T, y, x) en ligne 10. Cet appel incrémente $\text{degré}[x]$ mais laisse $\text{degré}[y]$ à d . Comme le nœud y n'est plus une racine, le pointeur pointant vers lui est supprimé du tableau A en ligne 11. Comme l'appel de RELIER-TAS-FIB incrémente la valeur de $\text{degré}[x]$, la ligne 12 restaure donc l'invariant selon lequel $d = \text{degré}[x]$.

Terminaison : On répète la boucle **tant que** jusqu'à ce que $A[d] = \text{NIL}$, auquel cas il n'y a aucune autre racine qui ait le même degré que x .

Après la fin de la boucle **tant que**, on affecte à $A[d]$ la valeur x en ligne 13 et l'on fait l'itération suivante de la boucle **pour**.

Les figures 20.3(c)–(e) montrent le tableau A et les arbres résultants après les trois premières itérations de la boucle **pour** des lignes 3–13. Lors de l'itération suivante de la boucle **pour**, il y a création de trois liaisons ; leurs résultats sont montrés sur les figures 20.3(f)–(h). Les figures 20.3(i)–(l) montrent le résultat des quatre itérations suivantes de la boucle **pour**.

Tout ce qui reste à faire maintenant, c'est le nettoyage. Une fois que la boucle **pour** des lignes 3–13 a pris fin, la ligne 14 vide la liste de racines et les lignes 15–19 reconstruisent cette liste à partir du tableau A . Le tas de Fibonacci résultant est montré à la figure 20.3(m). Après consolidation de la liste de racines, EXTRAIRE-MIN-TAS-FIB se termine en décrémentant $n[T]$ en ligne 11 et en retournant un pointeur vers le nœud supprimé z en ligne 12.

Observez que, si tous les arbres du tas de Fibonacci sont des arbres binomiaux non ordonnés avant l'exécution de EXTRAIRE-MIN-TAS-FIB, ils le restent après. Les arbres peuvent être modifiés de deux manières. Primo, aux lignes 3–5 de EXTRAIRE-MIN-TAS-FIB, chaque enfant x de la racine z devient une racine. D'après l'exercice 20.2.2, chaque nouvel arbre est lui-même un arbre binomial non ordonné. Secundo, les arbres ne sont reliés par RELIER-TAS-FIB que s'ils ont le même degré. Comme tous les arbres sont des arbres binomiaux non ordonnés avant la liaison, deux arbres dont les racines ont chacune k enfants doivent avoir la structure de U_k . L'arbre résultant a donc la structure de U_{k+1} .

On peut maintenant montrer que le coût amorti de l'extraction du nœud minimal d'un tas de Fibonacci à n nœuds est $O(D(n))$. Soit T le tas de Fibonacci juste avant l'opération EXTRAIRE-MIN-TAS-FIB.

Voici comment on peut évaluer le coût réel de l'extraction du nœud minimal. Il y a une contribution $O(D(n))$ qui vient de ce qu'il y a au plus $D(n)$ enfants du nœud minimal qui sont traités dans EXTRAIRE-MIN-TAS-FIB et du travail fait sur les lignes 1–2 et 14–19 de CONSOLIDER. Reste à analyser la contribution de la boucle **pour** des lignes 3–13. La taille de la liste de racines, lors de l'appel de CONSOLIDER, est

d'au plus $D(n) + t(T) - 1$, vu que la liste se compose des $t(T)$ nœuds originaux de la liste, moins le nœud racine extrait, plus les enfants du nœud extrait qui sont au plus $D(n)$. À chaque passage dans la boucle **tant que** des lignes 6–12, l'une des racines est reliée à une autre ; donc, le volume total de travail effectué dans la boucle **pour** est au plus proportionnel à $D(n) + t(T)$. Par conséquent, le travail réel total induit par l'extraction du nœud minimal est $O(D(n) + t(T))$.

Le potentiel avant extraction du nœud minimal est $t(T) + 2m(T)$; le potentiel après extraction est d'au plus $(D(n) + 1) + 2m(T)$, car il reste au plus $D(n) + 1$ racines et aucun nœud n'est marqué pendant l'opération. Le coût amorti est donc au maximum

$$\begin{aligned} & O(D(n) + t(T)) + ((D(n) + 1) + 2m(T)) - (t(T) + 2m(T)) \\ &= O(D(n)) + O(t(T)) - t(T) \\ &= O(D(n)), \end{aligned}$$

puisque l'on peut réévaluer les unités de potentiel de façon qu'elles dominent la constante cachée dans $O(t(T))$. Intuitivement, le coût de chaque liaison est compensé par la diminution de potentiel due au fait que la liaison diminue de un le nombre de racines. On verra à la section 20.4 que $D(n) = O(\lg n)$, de sorte que le coût amorti de l'extraction du nœud minimal est $O(\lg n)$.

Exercices

20.2.1 Montrer le tas de Fibonacci résultant d'un appel à EXTRAIRE-MIN-TAS-FIB sur le tas de Fibonacci de la figure 20.3(m).

20.2.2 Démontrer que le lemme 19.1 est valide pour les arbres binomiaux non triés, si l'on remplace la propriété 4 par la propriété 4'.

20.2.3 Montrer que, si les opérations de tas fusionnable sont les seules supportées, le degré maximal $D(n)$ d'un tas de Fibonacci à n nœuds vaut au plus $\lfloor \lg n \rfloor$.

20.2.4 Le professeur Blaise a élaboré une nouvelle structure de données basée sur les tas de Fibonacci. Un tas de Blaise possède la même structure qu'un tas de Fibonacci et supporte les opérations de tas fusionnable. Les implémentations des opérations sont les mêmes que pour les tas de Fibonacci, hormis que l'insertion et l'union font une consolidation comme dernière phase de leur travail. Quels sont les temps d'exécution des opérations de tas de Blaise, dans le cas le plus défavorable ? La structure de données du professeur est-elle vraiment inédite ?

20.2.5 Montrer que, quand les seules opérations admises sur les clés sont des comparaisons entre deux clés (comme c'est le cas pour toutes les implémentations proposées dans ce chapitre), il est impossible que toutes les opérations de tas fusionnable puissent s'exécuter en temps amorti $O(1)$.

20.3 DIMINUTION D'UNE CLÉ ET SUPPRESSION D'UN NŒUD

Dans cette section, nous allons voir comment de diminuer la clé d'un nœud dans un tas de Fibonacci dans un temps amorti $O(1)$ et comment supprimer un nœud d'un tas de Fibonacci à n nœuds dans un temps amorti $O(D(n))$. Ces opérations ne préservent pas la propriété selon laquelle tous les arbres d'un tas de Fibonacci sont des arbres binomiaux non triés. Néanmoins, ils en sont assez proches pour qu'on puisse borner le degré maximal $D(n)$ par $O(\lg n)$. Prouver l'existence de cette borne, chose que sera faite à la section 20.4, permet d'inférer que EXTRAIRE-MIN-TAS-FIB et SUPPRIMER-TAS-FIB s'exécutent dans un temps amorti $O(\lg n)$.

a) Diminution d'une clé

Dans le pseudo code suivant qui implémente l'opération DIMINUER-CLÉ-TAS-FIB, on suppose comme précédemment que supprimer un nœud d'une liste chaînée ne modifie aucun des champs contenus dans le nœud supprimé.

DIMINUER-CLÉ-TAS-FIB(T, x, k)

```

1  si  $k > clé[x]$ 
2    alors erreur « nouvelle clé plus grande que clé courante »
3   $clé[x] \leftarrow k$ 
4   $y \leftarrow p[x]$ 
5  si  $y \neq \text{NIL}$  et  $clé[x] < clé[y]$ 
6    alors COUPER( $T, x, y$ )
7          COUPE-EN-CASCADE( $T, y$ )
8  si  $clé[x] < clé[\min[T]]$ 
9    alors  $\min[T] \leftarrow x$ 

```

COUPER(T, x, y)

```

1  supprimer  $x$  de liste des enfant de  $y$ , en décrémentant  $degré[y]$ 
2  ajouter  $x$  à liste de racines de  $T$ 
3   $p[x] \leftarrow \text{NIL}$ 
4   $marqué[x] \leftarrow \text{FAUX}$ 

```

COUPE-EN-CASCADE(T, y)

```

1   $z \leftarrow p[y]$ 
2  si  $z \neq \text{NIL}$ 
3    alors si  $marqué[y] = \text{FAUX}$ 
4          alors  $marqué[y] \leftarrow \text{VRAI}$ 
5          sinon COUPER( $T, y, z$ )
6          COUPE-EN-CASCADE( $T, z$ )

```

La procédure DIMINUER-CLÉ-TAS-FIB fonctionne de la manière suivante. Les lignes 1–3 assurent que la nouvelle clé n'est pas plus grande que la clé courante de x , puis assignent la nouvelle clé à x . Si x est une racine, ou si $clé[x] \geq clé[y]$, alors y est

le parent de x et aucune modification de la structure n'est nécessaire, puisque l'ordre de tas min a été respecté. Les lignes 4–5 testent cette condition.

Si l'ordre de tas min n'a pas été respecté, de nombreux changements peuvent survenir. On commence par **détacher** x à la ligne 6. La procédure COUPER « coupe » le lien entre x et son parent y , ce qui fait de x une racine.

On utilise les champs *marqué* pour obtenir les bornes temporelles désirées. Ces champs enregistrent une petite partie de l'historique de chaque nœud. Supposons que x soit un nœud qui a vécu les événements suivants :

- 1) à une époque, x était une racine,
- 2) ensuite, x a été relié à un autre nœud,
- 3) enfin, deux enfant de x ont été supprimés via des coupes.

Dès qu'il a perdu son deuxième enfant, x est coupé de son parent, ce qui en fait une nouvelle racine. Le champ *marqué*[x] a la valeur VRAI si les étapes 1 et 2 ont eu lieu et si un enfant de x a été coupé. La procédure COUPER efface donc *marqué*[x] à la ligne 4, puisqu'elle exécute l'étape 1. (On peut voir à présent pourquoi la ligne 3 de RELIER-TAS-FIB efface *marqué*[y] : le nœud y est en train d'être relié à un autre nœud et l'étape 2 est donc en cours. La prochaine fois qu'un enfant de y sera coupé, *marqué*[y] prendra la valeur VRAI.)

Nous n'avons pas encore terminé, car x pourrait être le deuxième enfant coupé de son parent y depuis la dernière fois où y a été relié à un autre nœud. La ligne 7 de DIMINUER-CLÉ-TAS-FIB effectue donc une **coupe en cascade** sur y . Si y est une racine, alors le test à la ligne 2 de COUPE-EN-CASCADE permet à la procédure de rendre la main immédiatement. Si y n'est pas marqué, la procédure le marque à la ligne 4, puisque son premier enfant vient d'être coupé, puis rend la main. Toutefois, si y est marqué, c'est qu'il vient de perdre son deuxième enfant ; y est coupé à la ligne 5 et COUPE-EN-CASCADE s'appelle elle-même récursivement à la ligne 6 avec comme paramètre z , le parent de y . La procédure COUPE-EN-CASCADE remonte ainsi dans l'arbre, jusqu'à trouver soit une racine, soit un nœud non marqué.

Une fois que toutes les coupes en cascade ont eu lieu, les lignes 8–9 de DIMINUER-CLÉ-TAS-FIB terminent le traitement en mettant à jour *min*[T] si nécessaire. Le seul nœud dont la clé a changé était le nœud x dont on a diminué la clé. Par conséquent, le nouveau nœud minimal est soit le nœud minimal originel soit le nœud x .

La figure 20.4 montre l'exécution de deux appels à DIMINUER-CLÉ-TAS-FIB, à partir du tas de Fibonacci représenté à la figure 20.4(a). Le premier appel, montré à la figure 20.4(b), ne requiert aucune coupe en cascade. Le deuxième appel, montré aux figures 20.4(c)–(e), induit deux coupes en cascade.

Nous allons montrer à présent que le coût amorti de DIMINUER-CLÉ-TAS-FIB vaut seulement $O(1)$. On commence par déterminer son coût réel. La procédure DIMINUER-CLÉ-TAS-FIB prend un temps $O(1)$, plus le temps d'exécution des

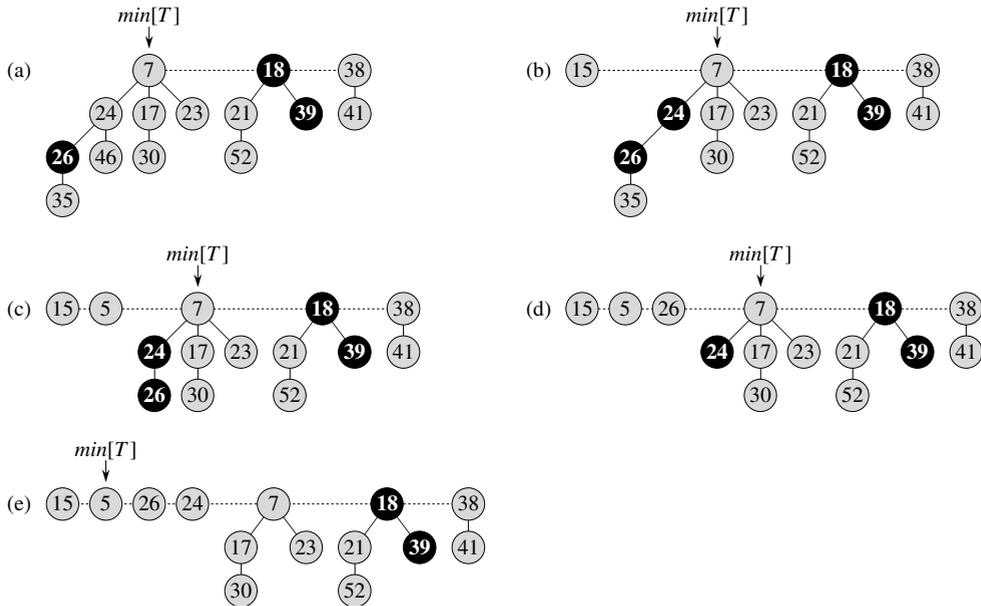


Figure 20.4 Deux appels à DIMINUER-CLÉ-TAS-FIB. (a) Le tas de Fibonacci initial. (b) Le nœud de clé 46 voit sa clé prendre la valeur 15. Le nœud devient une racine et son parent (de clé 24), qui avait précédemment perdu sa marque, est marqué à nouveau. (c)–(e) Le nœud de clé 35 voit sa clé diminuer pour prendre la valeur 5. Dans la partie (c), le nœud, contenant à présent la clé 5, devient racine. Son parent, le nœud de clé 26, est marqué, ce qui engendre une coupe en cascade. Le nœud de clé 26 est coupé de son parent et devient une racine non marquée (d). Une autre coupe survient, puisque le nœud de clé 24 est également marqué. Ce nœud est coupé de son parent, et devient une racine non marquée dans la partie (e). Les coupes en cascade s’arrêtent à cet endroit, puisque le nœud de clé 7 est une racine. (Même si le nœud n’avait pas été une racine, les coupes se seraient arrêtées, car il est non marqué.) Le résultat de l’opération DIMINUER-CLÉ-TAS-FIB est montré dans la partie (e), où $min[T]$ pointe sur le nouveau nœud minimal.

coupes en cascade. Supposons que COUPE-EN-CASCADE soit appelée récursivement c fois à partir d’un certain appel à DIMINUER-CLÉ-TAS-FIB. Chaque appel à COUPE-EN-CASCADE prend $O(1)$ si l’on ne compte pas les appels récursifs. Le coût réel de DIMINUER-CLÉ-TAS-FIB, en prenant en compte tous les appels récursifs, est donc $O(c)$.

On calcule ensuite la modification du potentiel. Soit T le tas de Fibonacci juste avant l’opération DIMINUER-CLÉ-TAS-FIB. Chaque appel récursif à COUPE-EN-CASCADE, excepté le dernier, coupe un nœud marqué et remet à zéro le bit de marquage. Après l’appel, on est en présence de $a(T) + c$ arbres (les $a(T)$ arbres initiaux, $c - 1$ arbres produits par les coupes en cascades et l’arbre de racine x) et au plus $m(T) - c + 2$ nœud marqués ($c - 1$ ont perdu leurs marques pendant les coupes en cascades et le dernier appel à COUPE-EN-CASCADE peut avoir marqué un nœud). Le changement de potentiel est donc au plus

$$((a(T) + c) + 2(m(T) - c + 2)) - (a(T) + 2m(T))) = 4 - c .$$

Donc, le coût amorti de DIMINUER-CLÉ-TAS-FIB vaut au plus

$$O(c) + 4 - c = O(1),$$

puisque l'on peut réévaluer les unités de potentiel pour dominer la constante cachée dans $O(c)$.

On peut maintenant voir pourquoi la définition de la fonction potentiel inclut un terme qui vaut deux fois le nombre de nœuds marqués. Lorsqu'un nœud y marqué est coupé lors d'une coupe en cascade, son bit de marquage est remis à zéro et le potentiel est réduit de 2 unités. Une unité de potentiel compense la coupe et l'effacement du marquage, et l'autre unité compense l'accroissement de potentiel dû à la transformation du nœud y en racine.

b) Suppression d'un nœud

La suppression d'un nœud d'un tas de Fibonacci à n nœuds est facile à effectuer dans un temps amorti $O(D(n))$, comme on le voit dans le pseudo code suivant. On suppose qu'il n'existe pas de valeur de clé égale à $-\infty$ dans le tas de Fibonacci.

```
SUPPRIMER-TAS-FIB( $T, x$ )
1  DIMINUER-CLÉ-TAS-FIB( $T, x, -\infty$ )
2  EXTRAIRE-MIN-TAS-FIB( $T$ )
```

SUPPRIMER-TAS-FIB est analogue à SUPPRIMER-TAS-BINOMIAL. Elle fait de x le nœud minimal du tas de Fibonacci en lui donnant une valeur de clé plus petite que toutes les autres, à savoir $-\infty$. Le nœud x est alors supprimé du tas de Fibonacci par la procédure EXTRAIRE-MIN-TAS-FIB. Le temps amorti pour SUPPRIMER-TAS-FIB est la somme du temps amorti $O(1)$ de DIMINUER-CLÉ-TAS-FIB et du temps amorti $O(D(n))$ de EXTRAIRE-MIN-TAS-FIB. Comme on verra à la section 20.4 que $D(n) = O(\lg n)$, le temps amorti de SUPPRIMER-TAS-FIB est $O(\lg n)$.

Exercices

20.3.1 On suppose qu'une racine x d'un tas de Fibonacci est marquée. Expliquer comment x a pu devenir une racine marquée. Montrer qu'il importe peu pour l'analyse que x soit marqué, même si ce n'est pas une racine qui avait été reliée à un autre nœud avant de perdre un enfant.

20.3.2 Justifier le temps amorti $O(1)$ de DIMINUER-CLÉ-TAS-FIB comme coût moyen par opération, en utilisant l'analyse par agrégat.

20.4 BORNE POUR LE DEGRÉ MAXIMAL

Pour démontrer que le temps amorti de EXTRAIRE-MIN-TAS-FIB et TAS-FIB-SUPPRIMER est $O(\lg n)$, on doit montrer que la borne supérieure $D(n)$ du degré d'un nœud d'un tas de Fibonacci à n nœuds vaut $O(\lg n)$. D'après l'exercice 20.2.3, quand tous les arbres du tas de Fibonacci sont des arbres binomiaux non triés, $D(n) = \lfloor \lg n \rfloor$. Toutefois, les coupes qui surviennent dans DIMINUER-CLÉ-TAS-FIB, peuvent obliger les arbres contenus dans le tas de Fibonacci à ne plus respecter les propriétés d'arbre binomial non trié. Dans cette section, nous allons montrer que le fait de couper un nœud de son parent dès qu'il perd deux enfant implique que $D(n) = O(\lg n)$. Plus précisément, nous démontrerons que $D(n) \leq \lfloor \log_\phi n \rfloor$, où $\phi = (1 + \sqrt{5})/2$.

Le principe de l'analyse est le suivant. Pour chaque nœud x du tas de Fibonacci, on définit $\text{taille}(x)$ comme étant le nombre de nœuds, y compris x lui-même, du sous-arbre enraciné en x . (Notez que x n'a pas besoin de faire partie de la liste de racines ; il peut être n'importe quel nœud.) Nous allons montrer que $\text{taille}(x)$ est exponentielle par rapport à $\text{degré}[x]$. Il faut garder à l'esprit que $\text{degré}[x]$ est constamment maintenu à jour en tant que degré de x .

Lemme 20.1 *Soit x un nœud d'un tas de Fibonacci, avec $\text{degré}[x] = k$. Soient y_1, y_2, \dots, y_k les enfant de x dans l'ordre dans lequel ils ont été reliés à x , en commençant par le plus ancien. Dans ce cas, $\text{degré}[y_1] \geq 0$ et $\text{degré}[y_i] \geq i - 2$ pour $i = 2, 3, \dots, k$.*

Démonstration : Manifestement, $\text{degré}[y_1] \geq 0$. Pour $i \geq 2$, on remarque qu'au moment où y_i a été relié à x , tous les y_1, y_2, \dots, y_{i-1} étaient des enfants de x ; on avait donc forcément $\text{degré}[x] = i - 1$. Le nœud y_i étant relié à x seulement si $\text{degré}[x] = \text{degré}[y_i]$, on avait donc aussi $\text{degré}[y_i] = i - 1$ à ce moment-là. Depuis lors, y_i a perdu au plus un enfant, puisqu'il aurait été coupé de x s'il en avait perdu deux. On en conclut que $\text{degré}[y_i] \geq i - 2$. \square

Nous arrivons enfin à la partie de l'analyse qui justifie le nom de « tas de Fibonacci ». On se souvient qu'à la section 3.2, pour $k = 0, 1, 2, \dots$, le k ème nombre de Fibonacci est défini par la récurrence

$$F_k = \begin{cases} 0 & \text{si } k = 0, \\ 1 & \text{si } k = 1, \\ F_{k-1} + F_{k-2} & \text{si } k \geq 2. \end{cases}$$

Le lemme suivant donne un autre moyen d'exprimer F_k .

Lemme 20.2 *Pour tous les entiers $k \geq 0$,*

$$F_{k+2} = 1 + \sum_{i=0}^k F_i.$$

Démonstration : La démonstration s'effectue par récurrence sur k . Quand $k = 0$,

$$\begin{aligned} 1 + \sum_{i=0}^0 F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2 . \end{aligned}$$

On admet maintenant l'hypothèse de récurrence $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, et l'on a

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i \right) \\ &= 1 + \sum_{i=0}^k F_i . \end{aligned}$$

□

Le lemme suivant et son corollaire complètent l'analyse. Ils font appel à l'inégalité (démontrée à l'exercice 3.2.7)

$$F_{k+2} \geq \phi^k ,$$

où ϕ est le nombre d'or, défini dans l'équation (3.22) par $\phi = (1 + \sqrt{5})/2 = 1,61803 \dots$

Lemme 20.3 Soit x un nœud d'un tas de Fibonacci, avec $k = \text{degré}[x]$. Alors, $\text{taille}(x) \geq F_{k+2} \geq \phi^k$, où $\phi = (1 + \sqrt{5})/2$.

Démonstration : Soit s_k la valeur minimale de $\text{taille}(z)$ sur tous les nœuds z tels que $\text{degré}[z] = k$. Trivialement, $s_0 = 1$, $s_1 = 2$ et $s_2 = 3$. Le nombre s_k vaut au plus $\text{taille}(x)$, et visiblement, la valeur de s_k croît de façon monotone avec k . Comme pour le lemme 20.1, soient y_1, y_2, \dots, y_k les enfants de x , dans l'ordre dans lequel ils ont été reliés à x . Pour calculer un majorant pour $\text{taille}(x)$, on compte un pour x lui-même et un pour le premier enfant y_1 (pour lequel $\text{taille}(y_1) \geq 1$), ce qui donne

$$\begin{aligned} \text{taille}(x) &\geq s_k \\ &= 2 + \sum_{i=2}^k s_{\text{degré}[y_i]} \\ &\geq 2 + \sum_{i=2}^k s_{i-2} , \end{aligned}$$

La dernière ligne découle du lemme 20.1 (qui fait que $\text{degré}[y_i] \geq i - 2$) et de la monotonie de s_k (qui fait que $s_{\text{degré}[y_i]} \geq s_{i-2}$). Nous montrons ensuite par récurrence sur k que $s_k \geq F_{k+2}$ pour tout entier k positif ou nul. Les bases, pour $k = 0$ et $k = 1$, sont triviales. Pour l'étape inductive, on suppose que $k \geq 2$ et que $s_i \geq F_{i+2}$ pour $i = 0, 1, \dots, k - 1$.

$$s_k \geq 2 + \sum_{i=2}^k s_{i-2} \geq 2 + \sum_{i=2}^k F_i = 1 + \sum_{i=0}^k F_i = F_{k+2} \quad (\text{d'après le lemme 20.2}) .$$

Nous avons donc démontré que $\text{taille}(x) \geq s_k \geq F_{k+2} \geq \phi^k$. □

Corollaire 20.4 *Le degré maximal $D(n)$ d'un nœud d'un tas de Fibonacci à n nœuds est $O(\lg n)$.*

Démonstration : Soit x un nœud d'un tas de Fibonacci à n nœuds, et soit $k = \text{degré}[x]$. D'après le lemme 20.3, on a $n \geq \text{taille}(x) \geq \phi^k$. En prenant les logarithmes de base ϕ , on obtient $k \leq \log_{\phi} n$. (En fait, comme k est un entier, $k \leq \lfloor \log_{\phi} n \rfloor$.) Le degré maximal $D(n)$ d'un nœud vaut donc $O(\lg n)$. □

Exercices

20.4.1 Le professeur Pinocchio affirme que la hauteur d'un tas de Fibonacci à n nœuds est $O(\lg n)$. Montrer que le professeur se trompe en trouvant, pour un entier positif n quelconque, une séquence d'opérations de tas de Fibonacci qui crée un tas de Fibonacci constitué d'un seul arbre qui est une chaîne linéaire de n nœuds.

20.4.2 Supposons qu'on généralise la règle de la coupe en cascade, de telle façon qu'un nœud x soit coupé de son parent dès lors qu'il perd son k ème enfant, pour une certaine constante entière k . (La règle de la section 20.3 utilise $k = 2$.) Pour quelles valeurs de k a-t-on $D(n) = O(\lg n)$?

PROBLÈMES

20.1. Autre implémentation de la suppression

Le professeur Pisano a proposé la variante suivante pour la procédure SUPPRIMER-TAS-FIB, en affirmant qu'elle s'exécute plus vite lorsque le nœud en cours de suppression n'est pas le nœud pointé par $\text{min}[T]$.

SUPPRIMER-PISANO(T, x)

```

1  si  $x = \text{min}[T]$ 
2    alors EXTRAIRE-MIN-TAS-FIB( $T$ )
3    sinon  $y \leftarrow p[x]$ 
4        si  $y \neq \text{NIL}$ 
5            alors COUPER( $T, x, y$ )
6                COUPE-EN-CASCADE( $T, y$ )
7            ajouter liste des enfant de  $x$  à liste de racines de  $T$ 
8            supprimer  $x$  de la liste de racines de  $T$ 
```

- a. L'affirmation du professeur selon laquelle cette procédure s'exécute plus vite est basée en partie sur l'hypothèse que la ligne 7 peut être exécutée dans un temps réel $O(1)$. Où est la faille dans cette hypothèse ?
- b. Donner un bon majorant pour le temps réel d'exécution de SUPPRIMER-PISANO quand $x \neq \min[T]$. Votre borne devra être fonction de $\text{degré}[x]$ et du nombre c d'appels à COUPE-EN-CASCADE.
- c. Soit T' le tas de Fibonacci résultant d'une exécution de SUPPRIMER-PISANO(T, x). En supposant que le nœud x n'est pas une racine, borner le potentiel de T' en fonction de $\text{degré}[x]$, c , $a(T)$ et $m(T)$.
- d. Conclure que le temps amorti de SUPPRIMER-PISANO n'est asymptotiquement pas meilleur que celui de SUPPRIMER-TAS-FIB, même quand $x \neq \min[T]$.

20.2. D'autres opérations sur les tas de Fibonacci

On souhaite étendre un tas de Fibonacci T pour lui permettre de supporter deux nouvelles opérations sans changer le temps d'exécution amorti des autres opérations.

- a. Donner une implémentation efficace de TAS-FIB-MODIFIER-CLÉ(T, x, k), procédure qui donne à la clé du nœud x la valeur k . Analyser le temps d'exécution amorti de votre implémentation selon que k est supérieur, inférieur ou égal à $\text{clé}[x]$.
- b. Donner une implémentation efficace de ELAGUER-TAS-FIB(T, r), qui supprime $\min(r, n[T])$ nœuds de T . Les nœuds à supprimer devront être arbitraires. Analyser le temps amorti d'exécution de votre implémentation (*Conseil* : On pourra avoir besoin de modifier la structure de données et la fonction potentiel.)

NOTES

Les tas de Fibonacci ont été introduits par Fredman et Tarjan [98]. Leur article décrit également l'application des tas de Fibonacci aux problèmes des plus courts chemins à origine unique, des plus courts chemins pour toutes paires, du couplage bipartite pondéré et de l'arbre couvrant minimal.

Ensuite, Driscoll, Gabow, Shrairman et Tarjan [81] ont développé « les tas relâchés » comme alternative aux tas de Fibonacci. Il existe deux variétés de tas relâchés. L'une donne les mêmes bornes temporelles amorties que les tas de Fibonacci. L'autre permet à DIMINUER-CLÉ de s'exécuter en temps (non amorti) $O(1)$ dans le cas le plus défavorable, et à EXTRAIRE-MIN et SUPPRIMER de s'exécuter en temps $O(\lg n)$ dans le cas le plus défavorable. Les tas relâchés sont également plus avantageux que les tas de Fibonacci pour les algorithmes parallèles.

Reportez-vous au notes du chapitre 6 pour connaître d'autres structures de données qui permettent des opérations DIMINUER-CLÉ rapides quand la suite des valeurs retournées par des appels EXTRAIRE-MIN croît de façon monotone avec le temps et que les données sont des entiers appartenant à un intervalle spécifique.

Chapitre 21

Structures de données pour ensembles disjoints

Certaines applications imposent de regrouper n éléments distincts dans une collection d'ensembles disjoints. Il est alors important de savoir à quel ensemble appartient un élément donné et de pouvoir réunir deux ensembles. Ce chapitre étudie des méthodes permettant de gérer une structure de données qui supporte ces opérations.

La section 21.1 décrit les opérations supportées par une structure de données pour ensembles disjoints et présente une application simple. Dans la section 21.2, on s'intéresse à une implémentation simple d'ensembles disjoints basée sur une liste chaînée. Une représentation plus efficace utilisant des arbres enracinés est donnée à la section 21.3. Le temps d'exécution de la représentation arborescente est linéaire dans tous les cas pratiques, bien qu'en théorie il soit supra linéaire. La section 21.4 définit et étudie une fonction qui croît très vite, ainsi que la fonction inverse qui croît très lentement et qui apparaît dans le temps d'exécution des opérations de l'implémentation basée sur les arbres ; la section utilise ensuite l'analyse amortie pour établir un majorant du temps d'exécution qui est tout juste supra linéaire.

21.1 OPÉRATIONS SUR LES ENSEMBLES DISJOINTS

Une *structure de données d'ensembles disjoints* gère une collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ d'ensembles dynamiques disjoints. Chaque ensemble est identifié par un *représentant*, qui est un certain membre de l'ensemble. Dans certaines applications, peu importe quel est l'élément qui fait office de représentant ; tout ce que l'on veut, c'est

que deux interrogations successives du représentant d'un ensemble dynamique qui n'a pas été modifié entre les deux requêtes donnent la même réponse. Dans d'autres applications, on peut spécifier à l'avance une règle pour le choix du représentant d'un ensemble ; on peut, par exemple, opter pour le plus petit élément de l'ensemble (au cas où les éléments peuvent être ordonnés.)

Comme dans les autres implémentations d'ensembles dynamiques que nous avons étudiées, chaque élément d'un ensemble est représenté par un objet. Soit x un objet quelconque, on souhaite supporter les opérations suivantes :

CRÉER-ENSEMBLE(x) est une procédure qui crée un nouvel ensemble dont le seul membre (et donc le représentant) est x . Comme les ensembles sont disjoints, il faut que x ne soit pas déjà membre d'un autre ensemble.

UNION(x, y) réunit les ensembles dynamiques qui contiennent x et y , appelons-les S_x et S_y , dans un nouvel ensemble qui est l'union de ces deux ensembles. Les deux ensembles sont supposés être disjoints avant l'opération. Le représentant de l'ensemble résultant est un membre quelconque de $S_x \cup S_y$, bien que de nombreuses implémentations de UNION choisissent le représentant de S_x ou de S_y comme nouveau représentant. Comme les ensembles de la collection doivent être disjoints, on « détruit » les ensembles S_x et S_y en les supprimant de la collection \mathcal{S} .

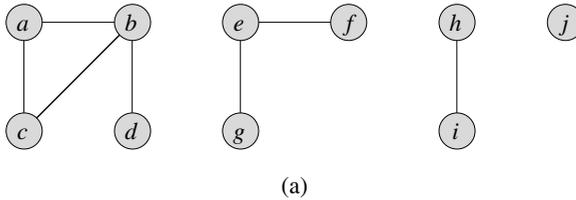
TROUVER-ENSEMBLE(x) retourne un pointeur vers le représentant de l'ensemble (unique) contenant x .

Dans tout ce chapitre, l'analyse du temps d'exécution des structures de données d'ensembles disjoints sera fonction de deux paramètres : n , nombre d'opérations CRÉER-ENSEMBLE et m , nombre total d'opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE. Comme les ensembles sont disjoints, chaque opération UNION diminue d'une unité le nombre des ensembles. Après $n-1$ opérations UNION, il ne reste donc plus qu'un seul ensemble. Le nombre d'opérations UNION est donc au plus égal à $n-1$. Notez également que, puisque les opérations CRÉER-ENSEMBLE sont incluses dans le nombre total d'opérations m , on a $m \geq n$. On supposera que les n opérations CRÉER-ENSEMBLE sont les n premières opérations effectuées.

a) Une application des structures de données d'ensembles disjoints

L'une des nombreuses applications des structures de données d'ensembles disjoints apparaît lorsqu'il s'agit de déterminer les composantes connexes d'un graphe non orienté (voir section B.4). La figure 21.1(a), par exemple, montre un graphe à quatre composantes connexes.

La procédure COMPOSANTES-CONNEXES qui suit utilise les opérations d'ensembles disjoints pour calculer les composantes connexes d'un graphe. Après un pré traitement basé sur COMPOSANTES-CONNEXES, la procédure MÊME-COMPOSANTE indique si deux sommets se trouvent dans la même composante



Arêtes traitées	Collection d'ensemble disjoints									
ensembles initiaux	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

connexe ⁽¹⁾. (L'ensemble des sommets d'un graphe G est noté $S[G]$, et l'ensemble des arêtes est noté $A[G]$.)

COMPOSANTES-CONNEXES(G)

```

1  pour chaque sommet  $v \in S[G]$ 
2    faire CRÉER-ENSEMBLE( $v$ )
3    pour chaque arête  $(u, v) \in A[G]$ 
4      faire si TROUVER-ENSEMBLE( $u$ )  $\neq$  TROUVER-ENSEMBLE( $v$ )
5        alors UNION( $u, v$ )

```

MÊME-COMPOSANTE(u, v)

```

1  si TROUVER-ENSEMBLE( $u$ ) = TROUVER-ENSEMBLE( $v$ )
2    alors retourner VRAI
3    sinon retourner FAUX

```

La procédure COMPOSANTES-CONNEXES place initialement chaque sommet v dans son propre singleton. Puis, pour chaque arête (u, v) , elle réunit les ensembles contenant u et v . D'après l'exercice 21.1.2, quand toutes les arêtes ont été traitées, deux sommets se trouvent dans la même composante connexe si et seulement si les

(1) Quand les arêtes du graphe sont « statiques » (jamais modifiées au cours du temps), les composantes connexes peuvent être calculées plus rapidement par une recherche « en profondeur d'abord » (exercice 22.3.11). Il peut arriver, néanmoins, que les arêtes soient ajoutées « dynamiquement » et qu'il faille mettre à jour les composantes connexes pour chaque ajout d'arête. Dans ce cas, l'implémentation donnée ici peut être plus efficace que l'exécution d'une nouvelle recherche en-profondeur-d'abord pour chaque nouvelle arête.

objets correspondants se trouvent dans le même ensemble. Donc, COMPOSANTES-CONNEXES calcule les ensembles de telle manière que MÊME-COMPOSANTE puisse déterminer si deux sommets appartiennent à la même composante connexe. La figure 21.1(b) montre la façon dont les ensembles disjoints sont calculés par COMPOSANTES-CONNEXES.

Dans une implémentation concrète de cet algorithme de composantes connexes, les représentations du graphe et de la structure de données d'ensembles disjoints devraient se référencer mutuellement. En clair, un objet représentant un sommet contiendrait un pointeur vers l'objet correspondant dans les ensembles disjoints, et *vice-versa*. Comme ces détails de programmation dépendent du langage employé, nous les ignorerons.

Exercices

21.1.1 Supposez que la procédure COMPOSANTES-CONNEXES soit exécutée sur le graphe non orienté $G = (S, A)$, où $S = \{a, b, c, d, e, f, g, h, i, j, k\}$ et que les arêtes de A soient traitées dans l'ordre suivant : $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Énumérer les sommets de chaque composante connexe après chaque itération des lignes 3–5.

21.1.2 Montrer que, après le traitement de toutes les arêtes par COMPOSANTES-CONNEXES, deux sommets se trouvent dans la même composante connexe si et seulement si ils appartiennent au même ensemble.

21.1.3 Pendant l'exécution de COMPOSANTES-CONNEXES sur un graphe non orienté $G = (S, A)$ à k composantes connexes, combien de fois TROUVER-ENSEMBLE est-elle appelée ? Combien de fois appelle-t-on UNION ? Les réponses seront exprimées en fonction de $|S|$, $|A|$ et k .

21.2 REPRÉSENTATION D'ENSEMBLES DISJOINTS PAR DES LISTES CHAÎNÉES

Une façon simple d'implémenter une structure de données d'ensembles disjoints consiste à représenter chaque ensemble par une liste chaînée. Le premier objet de chaque liste chaînée sert de représentant pour l'ensemble auquel il appartient. Chaque objet de la liste chaînée contient un élément de l'ensemble, un pointeur vers l'objet contenant l'élément suivant de l'ensemble et un pointeur vers le représentant de l'ensemble. Chaque liste gère des pointeurs *tête* et *queue*, pointant respectivement vers le représentant et vers le dernier objet de la liste. La figure 21.2(a) montre deux ensembles. À l'intérieur de chaque liste chaînée, les objets peuvent apparaître dans n'importe quel ordre (en conservant l'hypothèse selon laquelle le premier objet de chaque liste est le représentant de l'ensemble).

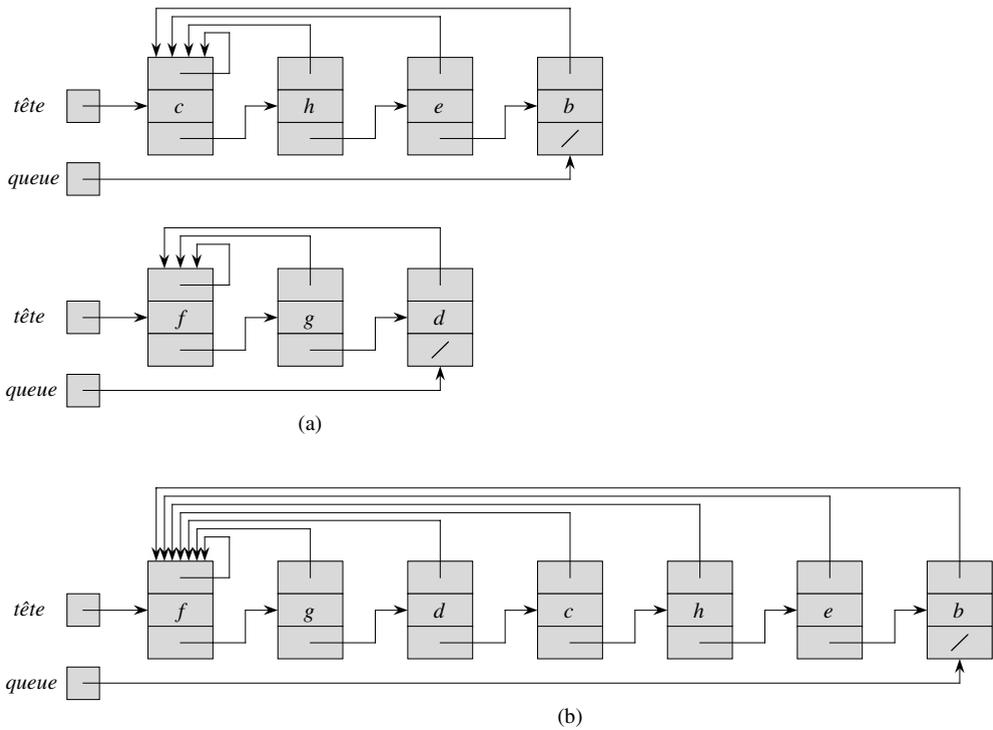


Figure 21.1 (a) Représentation par listes chaînées de deux ensembles. Le premier contient les objets *b*, *c*, *e* et *h* (avec *c* pour représentant) et le second contient les objets *d*, *f* et *g* (avec *f* comme représentant). Chaque objet de la liste et un pointeur vers le premier objet de la liste, qui est le représentant. Chaque liste gère des pointeurs *tête* et *queue*, pointant respectivement vers le représentant et vers le dernier objet de la liste. (b) Le résultat de $\text{UNION}(e, g)$. Le représentant de l'ensemble résultant est *f*.

Avec cette représentation par listes chaînées, CRÉER-ENSEMBLE et TROUVER-ENSEMBLE sont simples à implémenter et consomment un temps $O(1)$. Pour exécuter $\text{CRÉER-ENSEMBLE}(x)$, on crée une nouvelle liste chaînée dont l'objet unique est *x*. Pour $\text{TROUVER-ENSEMBLE}(x)$, on se contente de retourner le pointeur contenu dans *x* et pointant vers le représentant.

a) Une implémentation simple de l'union

L'implémentation la plus simple de l'opération UNION , qui utilise la représentation par listes chaînées, consomme beaucoup plus de temps que CRÉER-ENSEMBLE ou TROUVER-ENSEMBLE . Comme le montre la figure 21.2(b), $\text{UNION}(x, y)$ est exécutée par concaténation de la liste de *x* à la fin de la liste de *y*. On utilise le pointeur *queue* de la liste de *y* pour trouver rapidement l'emplacement où il faut ajouter la liste de *x*. Le représentant du nouvel ensemble est l'élément qui était au départ le représentant de l'ensemble contenant *y*. Malheureusement, pour chaque objet qui se trouvait

initialement dans la liste de x , il faut mettre à jour le pointeur vers le représentant, ce qui prend un temps linéaire en fonction de la longueur de la liste de x .

Opération	Nombre d'objets mis à jour
CRÉER-ENSEMBLE(x_1)	1
CRÉER-ENSEMBLE(x_2)	1
\vdots	\vdots
CRÉER-ENSEMBLE(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
\vdots	\vdots
UNION(x_{n-1}, x_n)	$n - 1$

Figure 21.2 Une séquence de $2n - 1$ opérations sur n objets, qui demande un temps $\Theta(n^2)$, soit $\Theta(n)$ en moyenne par opération, quand on utilise la représentation par listes chaînées et l'implémentation simple de UNION.

En fait, il n'est pas difficile de trouver une séquence de m opérations sur n objets qui demande un temps d'exécution $\Theta(n^2)$. Supposez que l'on ait les objets x_1, x_2, \dots, x_n . On exécute n opérations CRÉER-ENSEMBLE, suivies de $n - 1$ opérations UNION (voir figure 21.3, de sorte que $m = 2n - 1$. On dépense un temps $\Theta(n)$ pour faire les n opérations CRÉER-ENSEMBLE. Comme la i ème opération UNION met à jour i objets, le nombre total d'objets mis à jour par l'ensemble des $n - 1$ opérations UNION est

$$\sum_{i=1}^{n-1} i = \Theta(n^2).$$

Le nombre total d'opérations est $2n - 1$, ce qui fait que chaque opération en moyenne requiert un temps $\Theta(n)$. En d'autres termes, le temps amorti d'une opération est $\Theta(n)$.

b) Une heuristique de l'union pondérée

Dans le cas le plus défavorable, l'implémentation précédente de la procédure UNION demande un temps moyen de $\Theta(n)$ par appel. En effet, il se peut que l'on ajoute une liste longue à une liste courte, auquel cas il faut mettre à jour, pour chaque membre de la liste la plus longue, le pointeur vers le représentant. Supposons, en revanche, que chaque liste contienne également la longueur de la liste (qui est facilement gérable) et que l'on concatène toujours la plus petite liste à la plus longue (en cas d'égalité entre les longueurs, le choix est arbitraire). Avec cette *heuristique de l'union pondérée* simple, une seule opération UNION peut prendre encore un temps $\Omega(n)$ si les deux

ensembles ont $\Omega(n)$ éléments. Toutefois, comme le montre le théorème suivant, une séquence de m opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE, parmi lesquelles il y a n opérations CRÉER-ENSEMBLE, prend un temps $O(m+n \lg n)$.

Théorème 21.1 *Avec la représentation par liste chaînée des ensembles disjoints et l'heuristique de l'union pondérée, une séquence de m opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE, parmi lesquelles il y a n opérations CRÉER-ENSEMBLE, consomme un temps $O(m + n \lg n)$.*

Démonstration : On commence par calculer, pour chaque objet d'un ensemble de taille n , un majorant du nombre de fois que son pointeur vers le représentant est mis à jour. Considérons un objet x particulier. On sait que, chaque fois que le pointeur de x vers son représentant a été mis à jour, c'est que x se trouvait au départ dans le plus petit des deux ensembles. À la première mise à jour du pointeur de x pointant vers le représentant, l'ensemble résultant devait donc contenir au moins 2 éléments. À la deuxième mise à jour, l'ensemble résultant devait contenir au moins 4 éléments. En continuant le raisonnement, on observe que, pour un $k \leq n$ quelconque, après $\lceil \lg k \rceil$ mises à jour du pointeur de x vers le représentant, l'ensemble résultant doit posséder au moins k éléments. Comme l'ensemble le plus grand contient au plus n éléments, le pointeur de chaque objet vers le représentant a été mis à jour au plus $\lceil \lg n \rceil$ fois au cours de toutes les opérations UNION. Il faut aussi tenir compte des mises à jour des pointeurs *tête* et *queue* et de la mise à jour de la longueur de la liste, ce qui prend seulement $\Theta(1)$ par opération UNION. Le temps total utilisé pour la mise à jour des n objets est donc $O(n \lg n)$.

Le temps requis pour exécuter la séquence complète des m opérations s'en déduit facilement. Chaque opération CRÉER-ENSEMBLE et TROUVER-ENSEMBLE prend un temps $O(1)$, et il en existe $O(m)$. La séquence entière consomme donc un temps $O(m + n \lg n)$. \square

Exercices

21.2.1 Écrire un pseudo code pour les procédures CRÉER-ENSEMBLE, TROUVER-ENSEMBLE et UNION utilisant la représentation par listes chaînées et l'heuristique d'union pondérée. On supposera que chaque objet x contient un attribut $rep[x]$ qui pointe vers le représentant de l'ensemble contenant x , et que chaque ensemble S a des attributs $tête[S]$, $queue[S]$ et $size[S]$ (égal à la longueur de la liste).

21.2.2 Montrer la structure de données qui résulte du programme suivant, ainsi que les réponses retournées par les opérations TROUVER-ENSEMBLE. Utiliser la représentation par listes chaînées et l'heuristique de l'union pondérée.

```

1  pour  $i \leftarrow 1$  à 16
2      faire CRÉER-ENSEMBLE( $x_i$ )
3      pour  $i \leftarrow 1$  à 15 par pas de 2
4          faire UNION( $x_i, x_{i+1}$ )

```

```

5  pour  $i \leftarrow 1$  à 13 par pas de 4
6      faire UNION( $x_i, x_{i+2}$ )
7          UNION( $x_1, x_5$ )
8          UNION( $x_{11}, x_{13}$ )
9          UNION( $x_1, x_{10}$ )
10         TROUVER-ENSEMBLE( $x_2$ )
11         TROUVER-ENSEMBLE( $x_9$ )

```

On supposera que, si les ensembles contenant x_i et x_j ont la même taille, l'opération UNION(x_i, x_j) ajoute la liste de x_j à celle de x_i .

21.2.3 Adapter la démonstration du théorème 21.1 pour obtenir des bornes temporelles amorties qui soient $O(1)$ pour les opérations CRÉER-ENSEMBLE et TROUVER-ENSEMBLE, et $O(\lg n)$ pour l'opération UNION, en utilisant la représentation par listes chaînées et l'heuristique de l'union pondérée.

21.2.4 Donner une borne asymptotique serrée pour le temps d'exécution de la séquence d'opérations de la figure 21.3, si l'on choisit une représentation par listes chaînées associée à l'heuristique de l'union pondérée.

21.2.5 Suggérer une modification simple à la procédure UNION pour la représentation par listes chaînées, qui évite de devoir gérer le pointeur *queue* qui pointe vers le dernier objet de chaque liste. Avec ou sans heuristique de l'union pondérée, votre modification ne doit pas altérer le temps d'exécution asymptotique de la procédure UNION. (*Conseil* : Au lieu d'ajouter une liste à une autre, raccordez-les.)

21.3 FORÊTS D'ENSEMBLES DISJOINTS

Dans une implémentation plus rapide des ensembles disjoints, on représente les ensembles par des arborescences, où chaque nœud contient un élément et où chaque arbre représente un ensemble. Dans une *forêt d'ensembles disjoints*, illustrée à la figure 21.4(a), chaque élément pointe uniquement sur son parent. La racine de chaque arborescence contient le représentant, et elle est son propre parent. Comme nous le verrons, bien que les algorithmes directs utilisant cette représentation ne soient pas plus rapides que ceux utilisant la représentation en listes chaînées, en introduisant deux heuristiques, dites « union par rang » et « compression de chemin », on peut obtenir la structure de données d'ensembles disjoints la plus rapide asymptotiquement qui soit connue à ce jour.

Les trois opérations d'ensembles disjoints agissent de la manière suivante. L'opération CRÉER-ENSEMBLE se contente de créer une arborescence à un seul nœud. L'opération TROUVER-ENSEMBLE suit les pointeurs de parent jusqu'à trouver la racine de l'arborescence. Les nœuds visités sur le chemin menant à la racine constituent le *chemin de découverte*. Une opération UNION, représentée sur la figure 21.4(b), force la racine d'une arborescence à pointer vers la racine de l'autre.

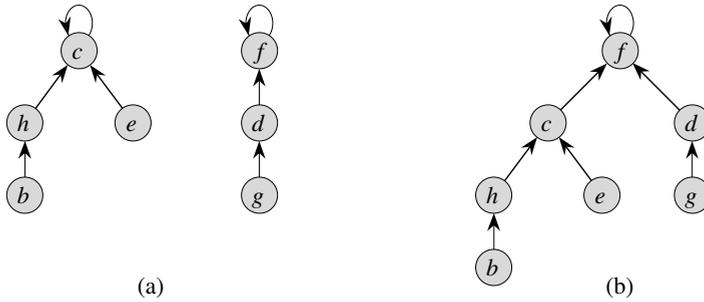


Figure 21.3 Une forêt d'ensembles disjoints. (a) Deux arborescences représentant les deux ensembles de la figure 21.1. L'arborescence de gauche représente l'ensemble $\{b, c, e, h\}$ (avec c comme représentant) et l'arborescence de droite représente l'ensemble $\{d, f, g\}$ (avec f comme représentant). (b) Le résultat de $\text{UNION}(e, g)$.

a) Heuristiques pour améliorer le temps d'exécution

Jusqu'à maintenant, nous n'avons pas fait mieux qu'avec la représentation par listes chaînées. Une séquence de $n - 1$ opérations UNION pourrait créer une arborescence qui se réduit à une chaîne linéaire de n nœuds. Toutefois, en utilisant deux heuristiques, on peut atteindre un temps d'exécution qui est quasi linéaire par rapport au nombre total d'opérations m .

La première heuristique, *l'union par rang*, ressemble à l'heuristique de l'union pondérée que nous avons utilisée avec la représentation par listes chaînées. Le principe est de faire pointer la racine de l'arborescence contenant le moins de nœuds vers celle de l'arborescence contenant le plus de nœuds. Au lieu de gérer explicitement la taille de la sous-arborescence associée à chaque nœud, nous utiliserons une approche qui facilite l'analyse. Pour chaque nœud, on gère un *rang* qui est un majorant de la hauteur du nœud. Dans l'union par rang, on fait pointer la racine de moindre rang vers celle de rang supérieur lors de l'opération UNION .

La seconde heuristique, *la compression de chemin*, est également très simple et très efficace. Comme le montre la figure 21.5, on s'en sert pendant les opérations TROUVER-ENSEMBLE pour faire pointer chaque nœud du chemin de découverte directement vers la racine. La compression de chemin ne modifie aucun rang.

b) Pseudo code pour forêts d'ensembles disjoints

Pour implémenter une forêt d'ensembles disjoints avec l'heuristique de l'union par rang, il faut gérer les rangs. Avec chaque nœud x , on gère la valeur entière $\text{rang}[x]$, qui est un majorant de la hauteur de x (nombre d'arcs sur le chemin le plus long entre x et une feuille descendante). Lorsqu'un singleton est créé par CRÉER-ENSEMBLE , le rang initial du nœud unique dans l'arborescence correspondante a la valeur 0. Chaque opération TROUVER-ENSEMBLE laisse tous les rangs inchangés. Quand UNION est appliquée à deux arborescences, deux cas se présentent selon que les racines des

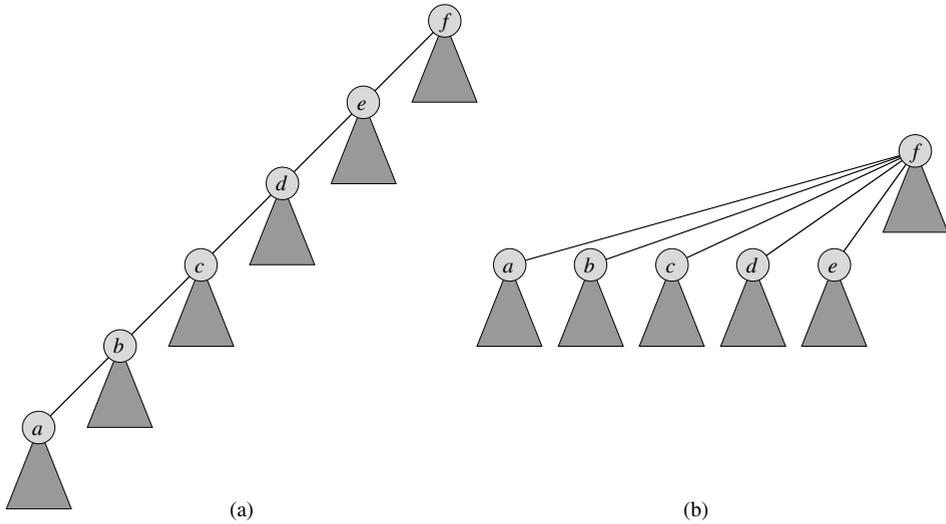


Figure 21.4 Compression de chemin lors de l'opération TROUVER-ENSEMBLE. Les flèches et les boucles aux racines sont omises. (a) Une arborescence représentant un ensemble avant l'exécution de TROUVER-ENSEMBLE(a). Les triangles représentent des sous-arborescences dont les racines sont les nœuds montrés sur la figure. Chaque nœud possède un pointeur vers son parent. (b) Le même ensemble après exécution de TROUVER-ENSEMBLE(a). À présent, chaque nœud du chemin de découverte pointe directement vers la racine.

deux arborescences ont ou non des rangs égaux. Si elles ont des rangs inégaux, la racine de plus haut rang devient le parent de la racine de moindre rang, mais les rangs eux-mêmes ne changent pas. En cas d'égalité entre les deux rangs, on choisit arbitrairement l'une des racines comme parent de l'autre et on incrémente son rang.

Traduisons cette méthode en pseudo code. On désigne le parent du nœud x par $p[x]$. La procédure LIER, sous-programme appelé par UNION, prend en entrée des pointeurs vers les deux racines.

CRÉER-ENSEMBLE(x)

- 1 $p[x] \leftarrow x$
- 2 $rang[x] \leftarrow 0$

UNION(x, y)

- 1 LIER(TROUVER-ENSEMBLE(x), TROUVER-ENSEMBLE(y))

LIER(x, y)

- 1 **si** $rang[x] > rang[y]$
- 2 **alors** $p[y] \leftarrow x$
- 3 **sinon** $p[x] \leftarrow y$
- 4 **si** $rang[x] = rang[y]$
- 5 **alors** $rang[y] \leftarrow rang[y] + 1$

La procédure TROUVER-ENSEMBLE avec compression de chemin est tout à fait simple.

TROUVER-ENSEMBLE(x)

```

1  si  $x \neq p[x]$ 
2    alors  $p[x] \leftarrow$  TROUVER-ENSEMBLE( $p[x]$ )
3    retourner  $p[x]$ 

```

On peut distinguer deux passes dans l'exécution de la procédure TROUVER-ENSEMBLE : la première remonte le chemin de découverte jusqu'à la racine, alors que la seconde redescend le chemin de découverte pour mettre à jour chaque nœud de manière qu'il pointe directement vers la racine. Chaque appel à TROUVER-ENSEMBLE(x) retourne $p[x]$ en ligne 3. Si x est la racine, la ligne 2 n'est pas exécutée et $p[x] = x$ est retourné. C'est le cas de terminaison de la récursivité se termine pour cause d'atteinte de la limite de la récursivité. Sinon, la ligne 2 est exécutée et l'appel récursif sur le paramètre $p[x]$ retourne un pointeur vers la racine. La ligne 2 met à jour le nœud x pour qu'il pointe directement vers la racine et ce pointeur est retourné en ligne 3.

c) Effet des heuristiques sur le temps d'exécution

L'union par rang et la compression de chemin améliorent chacune séparément le temps d'exécution des opérations de forêt d'ensembles disjoints et l'amélioration est encore plus grande si les deux heuristiques sont utilisées ensemble. À elle seule, l'union par rang engendre un temps d'exécution de $O(m \lg n)$ (voir exercice 21.4.4) et cette borne est serrée (voir exercice 21.3.3). Nous ne le démontrerons pas ici, mais s'il y a n opérations CRÉER-ENSEMBLE (et donc, au plus $n - 1$ opérations UNION) et f opérations TROUVER-ENSEMBLE, à elle seule l'heuristique de la compression de chemin donne un temps d'exécution $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$ dans le cas le plus défavorable.

Quand on utilise à la fois l'union par rang et la compression de chemin, le temps d'exécution dans le cas le plus défavorable est $O(m \alpha(n))$, où $\alpha(n)$ est une fonction à croissance *très* lente, qui sera définie à la section 21.4. Pour toutes les applications possibles et imaginables de la structure de données d'ensembles disjoints, $\alpha(n) \leq 4$; on peut donc considérer le temps d'exécution comme linéaire en m dans tous les cas pratiques. A la section 21.4, nous démontrerons ce majorant.

Exercices

21.3.1 Faire l'exercice 21.2.2 en utilisant une forêt d'ensembles disjoints avec union par rang et compression de chemin.

21.3.2 Écrire une version non récursive de TROUVER-ENSEMBLE avec compression de chemin.

21.3.3 Donner une séquence de m opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE, contenant n opérations CRÉER-ENSEMBLE, dont le temps d'exécution est $\Omega(m \lg n)$ quand on se sert uniquement de l'union par rang.

21.3.4 * Montrer qu'une séquence quelconque de m opérations CRÉER-ENSEMBLE, TROUVER-ENSEMBLE et LIER, où toutes les opérations LIER sont effectuées avant la première des opérations TROUVER-ENSEMBLE, s'exécute en $O(m)$ seulement, si à la fois la compression de chemin et l'union par rang sont utilisées. Que se passe-t-il si, dans la même situation, on n'a recours qu'à l'heuristique de compression de chemin ?

21.4^{*} ANALYSE DE L'UNION PAR RANG AVEC COMPRESSION DE CHEMIN

Comme indiqué à la section 21.3, le temps d'exécution des heuristiques combinées de l'union par rang et de la compression de chemin est $O(m \alpha(n))$ pour m opérations d'ensembles disjoints portant sur n éléments. Dans cette section, nous allons étudier la fonction α pour constater la lenteur avec laquelle elle croît. Puis, nous justifierons ce temps d'exécution en employant la méthode du potentiel pour l'analyse amortie.

a) *Une fonction à croissance très rapide et son inverse à croissance très lente*

Pour des entiers $k \geq 0$ et $j \geq 1$, on définit la fonction $A_k(j)$ par

$$A_k(j) = \begin{cases} j + 1 & \text{si } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{si } k \geq 1, \end{cases}$$

où l'expression $A_{k-1}^{(j+1)}(j)$ utilise la notation d'itération fonctionnelle donnée à la section 3.2. Plus précisément, $A_{k-1}^{(0)}(j) = j$ et $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ pour $i \geq 1$. À propos du paramètre k , nous parlerons de *niveau* de la fonction A .

La fonction $A_k(j)$ croît strictement par rapport à j et k . Rien que pour voir la rapidité de la croissance de cette fonction, commençons par établir des formules exactes donnant $A_1(j)$ et $A_2(j)$.

Lemme 21.2 *Pour tout entier $j \geq 1$, on a $A_1(j) = 2j + 1$.*

Démonstration : On commence par utiliser une récurrence sur i pour montrer que $A_0^{(i)}(j) = j + i$. Pour le cas initial, on a $A_0^{(0)}(j) = j = j + 0$. Comme hypothèse de récurrence, supposons que $A_0^{(i-1)}(j) = j + (i - 1)$. Alors $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i - 1)) + 1 = j + i$. Enfin, on note que $A_1(j) = A_0^{(j+1)}(j) = j + (j + 1) = 2j + 1$. \square

Lemme 21.3 *Pour tout entier $j \geq 1$, on a $A_2(j) = 2^{j+1}(j + 1) - 1$.*

Démonstration : On commence par utiliser une récurrence sur i pour montrer que $A_1^{(i)}(j) = 2^i(j + 1) - 1$. Pour le cas initial, on a $A_1^{(0)}(j) = j = 2^0(j + 1) - 1$.

Comme hypothèse de récurrence, supposons que $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$. Alors $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2 \cdot (2^{i-1}(j+1) - 1) + 1 = 2^i(j+1) - 2 + 1 = 2^i(j+1) - 1$. Enfin, on note que $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$. \square

Nous pouvons maintenant nous faire une idée de la rapidité de la croissance de $A_k(j)$, rien qu'en examinant $A_k(1)$ pour les niveaux $k = 0, 1, 2, 3, 4$. D'après la définition de $A_0(k)$ et les lemmes précédents, on a $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$, et $A_2(1) = 2^{1+1} \cdot (1 + 1) - 1 = 7$. On a aussi

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

et

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \\ &\gg A_2(2047) \\ &= 2^{2048} \cdot 2048 - 1 \\ &> 2^{2048} \\ &= (2^4)^{512} \\ &= 16^{512} \\ &\gg 10^{80}, \end{aligned}$$

qui est le nombre estimé d'atomes de l'univers observable.

On définit l'inverse de la fonction $A_k(n)$, pour un entier $n \geq 0$, par

$$\alpha(n) = \min \{k : A_k(1) \geq n\}.$$

En clair, $\alpha(n)$ est le plus petit niveau k pour lequel $A_k(1)$ est au moins égal à n . Des valeurs précédentes de $A_k(1)$, il ressort que

$$\alpha(n) = \begin{cases} 0 & \text{pour } 0 \leq n \leq 2, \\ 1 & \text{pour } n = 3, \\ 2 & \text{pour } 4 \leq n \leq 7, \\ 3 & \text{pour } 8 \leq n \leq 2047, \\ 4 & \text{pour } 2048 \leq n \leq A_4(1). \end{cases}$$

Ce n'est que pour des valeurs de n sortant de la norme (supérieures à $A_4(1)$, qui est un nombre colossal !) que $\alpha(n) > 4$; donc, $\alpha(n) \leq 4$ dans tous les cas de figure concrets.

b) Propriété des rangs

Dans le reste de cette section, nous démontrerons l'existence d'une borne $O(m \alpha(n))$ pour le temps d'exécution des opérations d'ensembles disjoints, quand on a recours à l'union par rang et à la compression de chemin. Pour établir cette borne, on commence par démontrer certaines propriétés simples des rangs.

Lemme 21.4 *Quel que soit le nœud x , on a $\text{rang}[x] \leq \text{rang}[p[x]]$, avec inégalité stricte si $x \neq p[x]$. La valeur de $\text{rang}[x]$ est initialement 0 et augmente au cours du temps jusqu'à ce qu'on ait $x \neq p[x]$; à partir de là, $\text{rang}[x]$ ne change plus. La valeur de $\text{rang}[p[x]]$ est une fonction temporelle monotone croissante.*

Démonstration : La démonstration est une récurrence directe sur le nombre d'opérations, qui utilise les implémentations de CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE étudiées à la section 21.3. Elle est laissée en exercice (voir exercice 21.4.1). \square

Corollaire 21.5 *Quand on suit le chemin menant d'un nœud quelconque à une racine, les rangs des nœuds croissent de manière stricte.*

Lemme 21.6 *Chaque nœud a un rang qui est au plus égal à $n - 1$.*

Démonstration : Le rang de chaque nœud commence à 0 et croît uniquement lors des opérations LIER. Comme il y a au plus $n - 1$ opérations UNION, il y a donc au plus $n - 1$ opérations LIER. Chaque opération LIER soit ne modifie aucun rang, soit augmente le rang d'un certain nœud de 1 ; donc, tous les rangs valent au plus $n - 1$. \square

Le lemme 21.6 fournit une borne lâche sur les rangs. En fait, chaque nœud a un rang au plus égal à $\lfloor \lg n \rfloor$ (voir exercice 21.4.2). La borne moins fine du lemme 21.6 suffira, cependant, à nos besoins.

c) Justification de la borne temporelle

Nous emploierons une analyse amortie basée sur la méthode du potentiel (voir section 17.3) pour prouver la borne $O(m \alpha(n))$. Pour faire l'analyse amortie, il est commode de partir du principe que l'on appelle l'opération LIER plutôt que l'opération UNION. En clair, comme les paramètres de la procédure LIER sont des pointeurs vers deux racines, on suppose que les opérations TROUVER-ENSEMBLE idoines sont faites séparément. Le lemme suivant montre que, si l'on compte les opérations TROUVER-ENSEMBLE supplémentaires induites par les appels à Union, le temps d'exécution asymptotique ne change pas.

Lemme 21.7 *Supposez que l'on convertisse une séquence S' de m' opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE en une séquence S de m opérations CRÉER-ENSEMBLE, LIER et TROUVER-ENSEMBLE, et ce en transformant chaque*

opération UNION en deux opérations TROUVER-ENSEMBLE suivies d'une opération LIER. Alors, si la séquence S est exécutée en temps $O(m\alpha(n))$, la séquence S' l'est en temps $O(m'\alpha(n))$.

Démonstration : Comme chaque opération UNION de la séquence S' est convertie en trois opérations de S , on a $m' \leq m \leq 3m'$. Comme $m = O(m')$, une borne $O(m\alpha(n))$ pour la séquence convertie S implique une borne $O(m'\alpha(n))$ pour la séquence originelle S' . \square

Dans le reste de cette section, on supposera que la séquence initiale de m' opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE a été convertie en une séquence de m opérations CRÉER-ENSEMBLE, LIER et TROUVER-ENSEMBLE. Nous allons maintenant prouver une borne temporelle $O(m\alpha(n))$ pour la séquence convertie et faire appel au lemme 21.7 pour justifier le temps d'exécution $O(m'\alpha(n))$ de la séquence originelle de m' opérations.

d) Fonction potentiel

La fonction potentiel que nous allons utiliser assigne un potentiel $\phi_q(x)$ à chaque nœud x de la forêt d'ensembles disjoints après q opérations. Le cumul des potentiels des nœuds donne le potentiel de la forêt complète : $\Phi_q = \sum_x \phi_q(x)$, où Φ_q désigne le potentiel de la forêt après q opérations. La forêt est vide avant la première opération et l'on définit arbitrairement $\Phi_0 = 0$. Il n'y aura jamais de potentiel Φ_q qui sera négatif.

La valeur de $\phi_q(x)$ dépend de ce que x est ou non une racine d'arborescence après la q ème opération. Si elle l'est, ou si $\text{rang}[x] = 0$, alors $\phi_q(x) = \alpha(n) \cdot \text{rang}[x]$.

Supposons maintenant que, après la q ème opération, x ne soit pas une racine et que $\text{rang}[x] \geq 1$. Il faut définir deux fonctions auxiliaires de x pour pouvoir définir $\phi_q(x)$. On définit d'abord

$$\text{niveau}(x) = \max \{k : \text{rang}[p[x]] \geq A_k(\text{rang}[x])\} .$$

En d'autres termes, $\text{niveau}(x)$ est le plus grand niveau k pour lequel A_k , appliqué au rang de x , n'est pas plus grand que le rang du parent de x .

Nous affirmons que

$$0 \leq \text{niveau}(x) < \alpha(n) , \tag{21.1}$$

que nous démontrons comme suit. On a

$$\begin{aligned} \text{rang}[p[x]] &\geq \text{rang}[x] + 1 \quad (\text{d'après le lemme 21.4}) \\ &= A_0(\text{rang}[x]) \quad (\text{by définition de } A_0(j)) , \end{aligned}$$

ce qui implique que $\text{niveau}(x) \geq 0$ et l'on a

$$\begin{aligned} A_{\alpha(n)}(\text{rang}[x]) &\geq A_{\alpha(n)}(1) \quad (\text{car } A_k(j) \text{ est strictement croissante}) \\ &\geq n \quad (\text{d'après la définition de } \alpha(n)) \\ &> \text{rang}[p[x]] \quad (\text{d'après le lemme 21.6}) , \end{aligned}$$

ce qui implique que $\text{niveau}(x) < \alpha(n)$. Notez que, comme $\text{rang}[p[x]]$ croît de façon monotone avec le temps, c'est aussi le cas de $\text{niveau}(x)$.

La seconde fonction auxiliaire est

$$\text{iter}(x) = \max \{i : \text{rang}[p[x]] \geq A_{\text{niveau}(x)}^{(i)}(\text{rang}[x])\} .$$

En clair, $\text{iter}(x)$ est le plus grand nombre de fois que l'on peut appliquer itérativement $A_{\text{niveau}(x)}$, appliquée initialement au rang de x , avant d'obtenir une valeur plus grande que le rang du parent de x .

Nous prétendons que

$$1 \leq \text{iter}(x) \leq \text{rang}[x] , \quad (21.2)$$

et nous allons le prouver comme suit. On a

$$\begin{aligned} \text{rang}[p[x]] &\geq A_{\text{niveau}(x)}(\text{rang}[x]) \quad (\text{d'après la définition de } \text{niveau}(x)) \\ &= A_{\text{niveau}(x)}^{(1)}(\text{rang}[x]) \quad (\text{d'après la définition de l'itération fonctionnelle}) , \end{aligned}$$

ce qui implique que $\text{iter}(x) \geq 1$; et l'on a

$$\begin{aligned} A_{\text{niveau}(x)}^{(\text{rang}[x]+1)}(\text{rang}[x]) &= A_{\text{niveau}(x)+1}(\text{rang}[x]) \quad (\text{d'après la définition de } A_k(j)) \\ &> \text{rang}[p[x]] \quad (\text{d'après la définition de } \text{niveau}(x)) , \end{aligned}$$

ce qui implique que $\text{iter}(x) \leq \text{rang}[x]$. Notez que, comme $\text{rang}[p[x]]$ croît de façon monotone avec le temps, pour que $\text{iter}(x)$ décroisse, il faut que $\text{niveau}(x)$ croisse. Tant que $\text{niveau}(x)$ ne change pas, $\text{iter}(x)$ croît ou ne change pas.

Ayant défini ces fonctions auxiliaires, nous pouvons définir le potentiel d'un nœud x après q opérations :

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rang}[x] & \text{si } x \text{ est une racine ou si } \text{rang}[x] = 0 , \\ (\alpha(n) - \text{niveau}(x)) \cdot \text{rang}[x] - \text{iter}(x) & \text{si } x \text{ n'est pas une racine et si } \text{rang}[x] \geq 1 . \end{cases}$$

Les deux lemmes suivants donnent des propriétés utiles concernant les potentiels de nœud.

Lemme 21.8 *Pour tout nœud x et pour toute valeur du compteur d'opérations q , on a*

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot \text{rang}[x] .$$

Démonstration : Si x est une racine ou si $\text{rang}[x] = 0$, alors $\phi_q(x) = \alpha(n) \cdot \text{rang}[x]$ par définition. Supposons maintenant que x ne soit pas une racine et que $\text{rang}[x] \geq 1$. On obtient un minorant de $\phi_q(x)$ en maximisant $\text{niveau}(x)$ et $\text{iter}(x)$. D'après la borne (21.1), $\text{niveau}(x) \leq \alpha(n) - 1$ et d'après la borne (21.2), $\text{iter}(x) \leq \text{rang}[x]$. Donc,

$$\begin{aligned} \phi_q(x) &\geq (\alpha(n) - (\alpha(n) - 1)) \cdot \text{rang}[x] - \text{rang}[x] \\ &= \text{rang}[x] - \text{rang}[x] \\ &= 0 . \end{aligned}$$

De même, on obtient un majorant de $\phi_q(x)$ en minimisant $\text{niveau}(x)$ et $\text{iter}(x)$. D'après la borne (21.1), $\text{niveau}(x) \geq 0$ et d'après la borne (21.2), $\text{iter}(x) \geq 1$. Donc,

$$\begin{aligned}\phi_q(x) &\leq (\alpha(n) - 0) \cdot \text{rang}[x] - 1 \\ &= \alpha(n) \cdot \text{rang}[x] - 1 \\ &< \alpha(n) \cdot \text{rang}[x].\end{aligned}$$

□

e) Modifications de potentiel et coûts amortis des opérations

Nous sommes prêts pour examiner la manière dont les opérations d'ensembles disjoints affectent les potentiels des nœuds. Sachant comment le potentiel change avec chaque opération, nous pourrions déterminer le coût amorti de chaque opération.

Lemme 21.9 *Soit x un nœud qui n'est pas une racine, et supposons que la q ème opération est LIER ou TROUVER-ENSEMBLE. Alors, après la q ème opération, $\phi_q(x) \leq \phi_{q-1}(x)$. En outre, si $\text{rang}[x] \geq 1$ et si $\text{niveau}(x)$ ou $\text{iter}(x)$ est modifiée par la q ème opération, alors $\phi_q(x) \leq \phi_{q-1}(x) - 1$. En clair, le potentiel de x ne peut pas croître et, si x a un rang positif et si $\text{niveau}(x)$ ou $\text{iter}(x)$ change, alors le potentiel de x diminue d'au moins 1.*

Démonstration : Comme x n'est pas une racine, la q ème opération ne modifie pas $\text{rang}[x]$; et comme n ne change pas après les n opérations CRÉER-ENSEMBLE initiales, $\alpha(n)$ ne change pas non plus. Donc, ces composantes de la formule du potentiel de x restent les mêmes après la q ème opération. Si $\text{rang}[x] = 0$, alors $\phi_q(x) = \phi_{q-1}(x) = 0$. Supposons maintenant que $\text{rang}[x] \geq 1$.

Rappelons-nous que $\text{niveau}(x)$ croît de façon monotone avec le temps. Si la q ème opération ne modifie pas $\text{niveau}(x)$, alors $\text{iter}(x)$ croît ou ne change pas. Si les deux fonctions $\text{niveau}(x)$ et $\text{iter}(x)$ ne changent pas, alors $\phi_q(x) = \phi_{q-1}(x)$. Si $\text{niveau}(x)$ ne change pas et que $\text{iter}(x)$ croît, alors elle croît d'au moins 1 et donc $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

Enfin, si la q ème opération accroît $\text{niveau}(x)$, elle l'accroît d'au moins 1, de sorte que la valeur du terme $(\alpha(n) - \text{niveau}(x)) \cdot \text{rang}[x]$ diminue d'au moins $\text{rang}[x]$. Comme $\text{niveau}(x)$ a augmenté, il se pourrait que la valeur de $\text{iter}(x)$ baisse; mais, selon la borne (21.2), la baisse est d'au plus $\text{rang}[x] - 1$. Donc, l'augmentation de potentiel due à la variation de $\text{iter}(x)$ est inférieure à la diminution de potentiel due à la variation de $\text{niveau}(x)$; on en conclut que $\phi_q(x) \leq \phi_{q-1}(x) - 1$. □

Nos trois derniers lemmes montrent que le coût amorti de chaque opération CRÉER-ENSEMBLE, LIER et TROUVER-ENSEMBLE est $O(\alpha(n))$. Rappelons-nous que, d'après l'équation (17.2), le coût amorti de chaque opération est égal à son coût réel plus l'augmentation de potentiel due à l'opération.

Lemme 21.10 *Le coût amorti de chaque opération CRÉER-ENSEMBLE est $O(1)$.*

Démonstration : Supposons que la q ème opération soit CRÉER-ENSEMBLE(x). Cette opération crée un nœud x de rang 0, de sorte que $\phi_q(x) = 0$. Il n'y a aucune autre modification de rang ni de potentiel, et donc $\Phi_q = \Phi_{q-1}$. Comme on sait déjà que le coût réel de l'opération CRÉER-ENSEMBLE est $O(1)$, la démonstration est terminée. □

Lemme 21.11 *Le coût amorti de chaque opération LIER est $O(\alpha(n))$.*

Démonstration : Supposons que la q ème opération soit LIER(x, y). Le coût réel de LIER est $O(1)$. Sans nuire à la généralité, supposons que l'opération LIER fasse de y le parent of x .

Pour calculer la variation de potentiel due à LIER, notons que les seuls nœuds dont les potentiels pourraient changer sont x, y et les enfants de y juste avant l'opération. Nous allons montrer que l'unique nœud dont le potentiel peut croître à cause de LIER est y , et que l'augmentation est d'au plus $\alpha(n)$:

- d'après le lemme 21.9, un nœud qui est un enfant de y juste avant LIER ne peut pas subir d'augmentation de potentiel due à LIER.
- d'après la définition de $\phi_q(x)$, on voit que, comme x était une racine juste avant la q ème opération, $\phi_{q-1}(x) = \alpha(n) \cdot \text{rang}[x]$. Si $\text{rang}[x] = 0$, alors $\phi_q(x) = \phi_{q-1}(x) = 0$. Sinon,

$$\begin{aligned} \phi_q(x) &= (\alpha(n) - \text{niveau}(x)) \cdot \text{rang}[x] - \text{iter}(x) \\ &< \alpha(n) \cdot \text{rang}[x] \quad (\text{d'après les inégalités (21.1) et (21.2)}) . \end{aligned}$$

Comme cette dernière quantité est $\phi_{q-1}(x)$, on voit que le potentiel de x diminue.

- Comme y est une racine avant l'opération LIER, $\phi_{q-1}(y) = \alpha(n) \cdot \text{rang}[y]$. Après l'opération LIER, y est encore une racine ; l'opération ne modifie pas le rang de y , ou bien elle l'accroît de 1. Donc, soit $\phi_q(y) = \phi_{q-1}(y)$ soit $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

L'augmentation de potentiel due à l'opération LIER est donc égale à $\alpha(n)$ au plus. Le coût amorti de l'opération LIER est $O(1) + \alpha(n) = O(\alpha(n))$. \square

Lemme 21.12 *Le coût amorti de chaque opération TROUVER-ENSEMBLE est $O(\alpha(n))$.*

Démonstration : Supposons que la q ème opération soit TROUVER-ENSEMBLE et que le chemin de découverte contienne s nœuds. Le coût réel de l'opération TROUVER-ENSEMBLE est $O(s)$. Nous allons montrer qu'aucun nœud ne voit son potentiel augmenter à la suite de TROUVER-ENSEMBLE et qu'il y a au moins $\max(0, s - (\alpha(n) + 2))$ nœuds du chemin de découverte qui voient leur potentiel décroître d'au moins 1.

Pour voir qu'aucun potentiel de nœud n'augmente, on commence par se référer au lemme 21.9 pour tous les nœuds autres que la racine. Si x est la racine, alors son potentiel est $\alpha(n) \cdot \text{rang}[x]$, qui ne change pas.

Montrons maintenant qu'il y a au moins $\max(0, s - (\alpha(n) + 2))$ nœuds qui voient leur potentiel décroître d'au moins 1. Soit x un nœud du chemin de découverte tel que : $\text{rang}[x] > 0$ et x est suivi, quelque part sur le chemin de découverte, d'un autre nœud y qui n'est pas une racine pour lequel $\text{niveau}(y) = \text{niveau}(x)$ juste avant l'opération TROUVER-ENSEMBLE. (Le nœud y n'est pas obligé de venir *immédiatement* après x sur le chemin de découverte.) Tous les nœuds du chemin de découverte, sauf au plus $\alpha(n) + 2$ d'entre eux, satisfont à ces contraintes concernant x . Ceux qui n'y satisfont pas sont le premier nœud du chemin (s'il a le rang 0), le dernier nœud du chemin (c'est-à-dire, la racine) et le dernier nœud w du chemin pour lequel $\text{niveau}(w) = k$, pour tout $k = 0, 1, 2, \dots, \alpha(n) - 1$.

Fixons un tel nœud x et montrons que son potentiel diminue d'au moins 1. Soit $k = \text{niveau}(x) = \text{niveau}(y)$. Juste avant la compression de chemin générée par TROUVER-ENSEMBLE, on a

$$\begin{aligned} \text{rang}[p[x]] &\geq A_k^{\text{iter}(x)}(\text{rang}[x]) && \text{(d'après la définition de iter}(x)) , \\ \text{rang}[p[y]] &\geq A_k(\text{rang}[y]) && \text{(d'après la définition de niveau}(y)) , \\ \text{rang}[y] &\geq \text{rang}[p[x]] && \text{(d'après le corollaire 21.5 et parce que} \\ &&& \text{y vient après } x \text{ sur le chemin de découverte) .} \end{aligned}$$

En combinant ces inégalités et en donnant à i la valeur de $\text{iter}(x)$ avant la compression de chemin, on a

$$\begin{aligned} \text{rang}[p[y]] &\geq A_k(\text{rang}[y]) \\ &\geq A_k(\text{rang}[p[x]]) && \text{(car } A_k(j) \text{ est strictement croissante)} \\ &\geq A_k(A_k^{\text{iter}(x)}(\text{rang}[x])) \\ &= A_k^{i+1}(\text{rang}[x]) . \end{aligned}$$

Comme la compression de chemin entraîne que x et y auront le même parent, on sait que, après la compression, $\text{rang}[p[x]] = \text{rang}[p[y]]$ et que la compression ne diminue pas $\text{rang}[p[y]]$. Comme $\text{rang}[x]$ ne change pas, après compression on a $\text{rang}[p[x]] \geq A_k^{i+1}(\text{rang}[x])$. Donc, la compression de chemin entraîne que $\text{iter}(x)$ augmente (au moins jusqu'à $i + 1$) ou que $\text{niveau}(x)$ augmente (ce qui se produit si $\text{iter}(x)$ augmente au moins jusqu'à $\text{rang}[x] + 1$). Dans l'un ou l'autre cas, d'après le lemme 21.9, on a $\phi_q(x) \leq \phi_{q-1}(x) - 1$. Donc, le potentiel de x diminue d'au moins 1. Le coût amorti de l'opération TROUVER-ENSEMBLE est égal au coût réel plus la variation de potentiel. Le coût réel est $O(s)$, et l'on a montré que le potentiel total diminue d'au moins $\max(0, s - (\alpha(n) + 2))$. Le coût amorti est donc au plus égal à $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, puisque l'on peut réévaluer les unités de potentiel pour dominer la constante cachée dans $O(s)$. \square

En regroupant les lemmes précédents, on obtient le théorème suivant.

Théorème 21.13 *Une séquence de m opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE, dont n d'entre elles sont des opérations CRÉER-ENSEMBLE, peut s'effectuer sur une forêt d'ensembles disjoints, avec union par rang et compression de chemin, en temps $O(m \alpha(n))$ dans le cas le plus défavorable.*

Démonstration : Immédiate d'après les lemmes 21.7, 21.10, 21.11 et 21.12. \square

Exercices

21.4.1 Démontrer le lemme 21.4.

21.4.2 Prouver que chaque nœud a un rang qui est au plus égal à $\lfloor \lg n \rfloor$.

21.4.3 à la lumière de l'exercice 21.4.2, combien de bits faut-il pour stocker $\text{rang}[x]$ pour chaque nœud x ?

21.4.4 En utilisant l'exercice 21.4.2, donner une démonstration simple qui prouve que les opérations sur une forêt d'ensembles disjoints, avec union par rang mais sans compression de chemin, s'exécutent en $O(m \lg n)$.

21.4.5 Le professeur Dante tient le raisonnement suivant : comme les rangs des nœuds augmentent de façon strictement croissante le long d'un chemin menant à la racine, les niveaux des nœuds doivent croître de façon monotone le long du chemin. En d'autres termes, si $\text{rang}(x) > 0$ et $p[x]$ n'est pas une racine, alors $\text{niveau}(x) \leq \text{niveau}(p[x])$. Le professeur raisonne-t-il sainement ?

21.4.6 * Soit la fonction $\alpha'(n) = \min \{k : A_k(1) \geq \lg(n+1)\}$. Montrer que $\alpha'(n) \leq 3$ pour toutes les valeurs réalistes de n et, en utilisant l'exercice 21.4.2, montrer comment modifier la démonstration de la fonction potentiel pour prouver qu'une séquence de m opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE, parmi lesquelles il y a n opérations CRÉER-ENSEMBLE, peut se faire sur une forêt d'ensembles disjoints, utilisant l'union par rang et la compression de chemin, en un temps $O(m \alpha'(n))$ dans le cas le plus défavorable.

PROBLÈMES

21.1. Minimum différé

Le *problème du minimum différé* propose de maintenir à jour un ensemble dynamique T d'éléments du domaine $\{1, 2, \dots, n\}$ pouvant subir les opérations INSÉRER et EXTRAIRE-MIN. Soit une séquence S de n appels à INSÉRER et m appels à EXTRAIRE-MIN, où chaque clé de $\{1, 2, \dots, n\}$ est insérée exactement une fois. On souhaite déterminer la clé retournée par chaque appel à EXTRAIRE-MIN. Plus précisément, il s'agit de remplir un tableau *extrait*[1..m], où pour $i = 1, 2, \dots, m$, *extrait*[i] est la clé retournée par le i ème appel à EXTRAIRE-MIN. Le problème est « différé » au sens où il est possible de traiter entièrement la séquence S avant de déterminer n'importe laquelle des clés retournées.

- a. Dans l'instance ci-dessous du problème du minimum différé, chaque opération INSÉRER est représentée par un nombre et chaque EXTRAIRE-MIN est représentée par la lettre E :

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5 .

Remplir le tableau *extrait* avec les valeurs adéquates.

Pour développer un algorithme résolvant ce problème, on découpe la séquence S en sous-séquences homogènes. Autrement dit, on représente S par

$$I_1, E, I_2, E, I_3, \dots, I_m, E, I_{m+1} ,$$

où chaque E représente un appel unique à EXTRAIRE-MIN et chaque I_j représente une séquence (éventuellement vide) d'appels à INSÉRER. Pour chaque sous-séquence I_j , on commence par placer les clés insérées par ces opérations dans un ensemble K_j , qui est vide si I_j est vide. Puis, on exécute la procédure suivante :

MINIMUM-DIFFÉRÉ(m, n)

```

1  pour  $i \leftarrow 1$  à  $n$ 
2      faire déterminer  $j$  tel que  $i \in K_j$ 
3          si  $j \neq m + 1$ 
4              alors  $extrait[j] \leftarrow i$ 
5                  soit  $l$  la plus petite valeur supérieure à  $j$ 
                    pour laquelle l'ensemble  $K_l$  existe,
6                   $K_l \leftarrow K_j \cup K_l$ , avec destruction de  $K_j$ 
7  retourner  $extrait$ 

```

- b. Dire pourquoi le tableau *extrait* retourné par MINIMUM-DIFFÉRÉ est correct.
- c. Comment peut-on utiliser une structure de données d'ensembles disjoints pour implémenter efficacement MINIMUM-DIFFÉRÉ. Donner une borne serrée pour le temps d'exécution, dans le cas le plus défavorable, de votre implémentation.

21.2. Recherche de la profondeur

Dans le problème de *recherche de la profondeur*, on s'intéresse au comportement d'une forêt $\mathcal{F} = \{T_i\}$ d'arborescences supportant trois opérations :

CRÉER-ARBRE(v) crée une arborescence dont le seul nœud est v .

TROUVER-PROFONDEUR(v) retourne la profondeur du nœud v à l'intérieur de son arbre.

GREFFER(r, v) fait du nœud r , dont on suppose qu'il est racine d'une arborescence, le fils d'un nœud v (racine ou non) dont on suppose qu'il se trouve dans une arborescence différente de r .

- a. On suppose que les arborescences sont représentées comme dans une forêt d'ensembles disjoints : $p[v]$ est le parent du nœud v , sauf si v est une racine, auquel cas $p[v] = v$. Si l'on implémente GREFFER(r, v) via l'affectation $p[r] \leftarrow v$ et si l'on implémente TROUVER-PROFONDEUR(v) en suivant le chemin de découverte jusqu'à la racine, en retournant le nombre de nœuds rencontrés autres que v , montrer que le temps d'exécution, dans le cas le plus défavorable, d'une séquence de m opérations CRÉER-ARBRE, TROUVER-PROFONDEUR et GREFFER est $\Theta(m^2)$.

En utilisant les heuristiques de l'union par rang et de la compression de chemin, on peut réduire le temps d'exécution dans le pire des cas. On a recours à une forêt d'ensembles disjoints $\mathcal{S} = \{S_i\}$, où chaque ensemble S_i (qui est lui-même une arborescence) correspond à une arborescence T_i de la forêt \mathcal{F} . Cependant, la structure des arborescences à l'intérieur d'un ensemble S_i ne correspond pas nécessairement à celle de T_i . En fait, l'implémentation de S_i ne reflète pas exactement la relation parent-enfant mais permet, néanmoins, de déterminer la profondeur d'un nœud dans T_i .

Le principe est de conserver dans chaque nœud v une « pseudo distance » $d[v]$, définie de telle sorte que la somme des pseudo distances le long du chemin reliant v à la racine de son ensemble S_i soit égale à la profondeur de v dans T_i . Autrement dit,

si le chemin entre v et sa racine dans S_i est v_0, v_1, \dots, v_k , où $v_0 = v$ et v_k est la racine de S_i , alors la profondeur de v dans T_i est égale à $\sum_{j=0}^k d[v_j]$.

- b. Donner une implémentation de CRÉER-ARBRE.
- c. Comment peut-on modifier TROUVER-ENSEMBLE pour implémenter TROUVER-PROFONDEUR ? Votre implémentation devra effectuer une compression de chemin et son temps d'exécution devra être linéaire par rapport à la longueur du chemin de découverte. Assurez-vous que votre implémentation met correctement à jour les pseudo distances.
- d. Comment faut-il modifier les procédures UNION et LIER pour implémenter GREFFER(r, v), qui combine les ensembles contenant r et v . Vérifiez que votre implémentation met correctement à jour les pseudo distances. Notez que la racine d'un ensemble S_i n'est pas nécessairement la racine de l'arbre T_i correspondant.
- e. Donner une borne serrée du temps d'exécution, dans le cas le plus défavorable, d'une séquence de m opérations CRÉER-ARBRE, TROUVER-PROFONDEUR et GREFFER, parmi lesquelles on trouve n opérations CRÉER-ARBRE.

21.3. Algorithme différé de Tarjan pour le premier ancêtre commun

Le *premier ancêtre commun* de deux nœuds u et v d'une arborescence T est le nœud w qui est à la fois ancêtre de u et v et qui a la plus grande profondeur dans T . Dans le problème différé *du premier ancêtre commun*, on dispose en entrée d'une arborescence T et d'un ensemble arbitraire $P = \{\{u, v\}\}$ de paires non ordonnées de nœuds de T et l'on souhaite déterminer le premier ancêtre commun pour chaque paire de P .

Pour résoudre le problème différé du premier ancêtre commun, la procédure suivante effectue un parcours de l'arbre T via l'appel initial PAC(*racine*[T]). On suppose que chaque nœud est colorié en BLANC avant le parcours.

PAC(u)

```

1  CRÉER-ENSEMBLE( $u$ )
2  ancêtre[TROUVER-ENSEMBLE( $u$ )] ←  $u$ 
3  pour chaque enfant  $v$  de  $u$  dans  $T$ 
4      faire PAC( $v$ )
5          UNION( $u, v$ )
6          ancêtre[TROUVER-ENSEMBLE( $u$ )] ←  $u$ 
7          couleur[ $u$ ] ← NOIR
8      pour chaque nœud  $v$  tel que  $\{u, v\} \in P$ 
9          faire si couleur[ $v$ ] = NOIR
10         alors afficher « Le premier ancêtre commun de »
                 $u$  « et »  $v$  « est » ancêtre[TROUVER-ENSEMBLE( $v$ )]

```

- a. Montrer que, pour chaque paire $\{u, v\} \in P$, la ligne 10 est exécutée exactement une fois.

- b. Montrer qu'au moment de l'appel $\text{PAC}(u)$, le nombre d'ensembles de la structure de données d'ensembles disjoints est égal à la profondeur de u dans T .
- c. Démontrer que PAC affiche correctement le premier ancêtre commun de u et v pour toute paire $\{u, v\} \in P$.
- d. Analyser le temps d'exécution de PAC, en supposant qu'on utilise l'implémentation de la structure de données d'ensembles disjoints présentée à la section 21.3.

NOTES

Beaucoup des résultats importants concernant les structures de données d'ensembles disjoints sont dus, au moins en partie, à R. E. Tarjan. En employant l'analyse par agrégat, Tarjan [290, 292] a donné la première borne supérieure exprimée à partir de l'inverse à croissance très lente $\hat{\alpha}(m, n)$ de la fonction d'Ackermann. (La fonction $A_k(j)$ donnée à la section 21.4 ressemble à la fonction d'Ackermann, et la fonction $\alpha(n)$ ressemble à l'inverse. Tant $\alpha(n)$ que $\hat{\alpha}(m, n)$ sont au plus égales à 4 pour toutes les valeurs réalistes de m et n .) Un majorant de $O(m \lg^* n)$ avait été précédemment donné par Hopcroft et Ullman [5, 155]. La section 21.4 s'inspire d'une récente analyse due à Tarjan [294], laquelle s'appuie sur une analyse due à Kozen [193]. Harfst et Reingold [139] donnent une version, basée sur la méthode du potentiel, de la première borne de Tarjan.

Tarjan et van Leeuwen [295] étudient des variantes de l'heuristique de la compression de chemin, et notamment des « méthodes une-passe » qui présentent parfois des facteurs constants plus performants que ceux des méthodes deux-passes. Comme c'était le cas avec les premières analyses de Tarjan concernant l'heuristique basique de la compression de chemin, les analyses de Tarjan et van Leeuwen utilisent la méthode de l'agrégat. Harfst et Reingold [139] ont montré ultérieurement comment modifier légèrement la fonction potentiel pour adapter à ces variantes une-passe leur analyse à compression de chemin. Gabow et Tarjan [103] montrent que, dans certaines applications, on peut exécuter les opérations d'ensembles disjoints en temps $O(m)$.

Tarjan [291] a montré qu'on ne pouvait pas échapper à un minorant temporel $\Omega(m \alpha(m, n))$ pour les opérations sur des structures de données d'ensembles disjoints satisfaisant à certaines conditions techniques. Ce minorant fut ensuite généralisé par Fredman et Saks [97] qui montrèrent que, dans le cas le plus défavorable, on avait besoin d'accéder à $\Omega(m \alpha(m, n))$ mots mémoire de $(\lg n)$ bits.

PARTIE 6

ALGORITHMES POUR LES GRAPHS

Les graphes sont une structure de données dont l'importance ne cesse de croître en informatique, et les algorithmes permettant de les manipuler sont donc fondamentaux pour l'informatique. Il existe des centaines de problèmes calculatoires intéressants qui sont définis en termes de graphes. Dans cette partie, nous nous arrêterons sur quelques-uns parmi les plus significatifs.

Le chapitre 22 montre comment on peut représenter un graphe dans un ordinateur, puis présente des algorithmes de parcours en largeur et en profondeur dans un groupe. Deux applications de la recherche en profondeur sont étudiées ici : le tri topologique d'un graphe orienté acyclique ; la décomposition d'un graphe orienté en ses composantes fortement connexes.

Le chapitre 23 décrit comment calculer un arbre couvrant de poids minimum d'un graphe. Un tel arbre est défini comme la façon la plus économique de connecter tous les sommets quand on associe un poids à chaque arête. Les algorithmes de calcul d'arbre couvrant minimum sont de bons exemples d'algorithmes gloutons (voir chapitre 16).

Les chapitres 24 et 25 traitent du problème de calcul des plus courts chemins entre des sommets quand chaque arc a une longueur (ou « poids ») associée. Le chapitre 24 se penche sur le calcul des plus courts chemins reliant une certaine origine à tous les autres sommets, alors que le chapitre 25 traite du calcul des plus courts chemins entre toutes les paires de sommets.

Enfin, le chapitre 26 montre comment calculer un flot maximal de produits dans un réseau (graphe orienté) qui a une source de produits donnée, une destination spécifiée et des capacités spécifiées concernant le volume de produits susceptible de circuler sur chaque arc. Ce problème général apparaît sous de nombreuses formes, et un bon algorithme de calcul des flots maximum peut permettre de résoudre efficacement toutes sortes de problèmes de cette nature.

Quand on décrit le temps d'exécution d'un algorithme sur un graphe $G = (S, A)$ donné, on mesure habituellement la taille de l'entrée en fonction du nombre de sommets $|S|$ et du nombre d'arcs $|A|$ du graphe. Autrement dit, pour décrire la taille de l'entrée, on fait appel à deux paramètres au lieu d'un seul. Nous adopterons pour ces paramètres une convention notationnelle courante. À l'intérieur d'une notation asymptotique (comme la notation O ou la notation Θ), et *seulement* à l'intérieur de la notation, le symbole S désignera $|S|$ et le symbole A désignera $|A|$. Par exemple, on pourra dire « l'algorithme s'exécute en $O(SA)$ » pour signifier que le temps d'exécution de l'algorithme est $O(|S| |A|)$. Cette convention facilite la lecture des formules de temps d'exécution, sans qu'il y ait risque d'ambiguïté.

Une autre convention apparaît dans le pseudo code. L'ensemble des sommets d'un graphe G est noté $S[G]$ et l'ensemble de ses arcs est noté $A[G]$. Autrement dit, le pseudo code considère l'ensemble des sommets et celui des arcs comme des attributs du graphe.

Chapitre 22

Algorithmes élémentaires pour les graphes

Ce chapitre introduit quelques méthodes de représentation et de parcours d'un graphe. Parcourir un graphe revient à emprunter de façon systématique les arcs du graphe, pour en visiter les sommets. Un algorithme de parcours permet de mettre en évidence plusieurs caractéristiques de la structure du graphe. De nombreux algorithmes utilisent le résultat d'un parcours pour obtenir ces informations structurelles sur le graphe d'entrée. D'autres sont simplement des adaptations des algorithmes élémentaires de parcours de graphes. Les techniques de parcours de graphe sont au cœur des algorithmes de graphes.

La section 22.1 étudie les deux représentations de graphe les plus courantes : les listes d'adjacences et les matrices d'adjacences. La section 22.2 présente un algorithme de parcours simple, le parcours en largeur, et montre comment créer une arborescence de parcours en largeur. La section 22.3 présente le parcours en profondeur, et démontre quelques résultats fondamentaux sur l'ordre de visite des sommets. La section 22.4 fournit notre première véritable application du parcours en profondeur : le tri topologique d'un graphe orienté sans circuit. Une deuxième application du parcours en profondeur, permettant de trouver les composantes fortement connexes d'un graphe orienté, est donnée à la section 22.5.

22.1 REPRÉSENTATION DES GRAPHES

Il existe deux façons classiques de représenter un graphe $G = (S, A)$: par un ensemble de listes d'adjacences, ou par une matrice d'adjacences. La représentation par listes d'adjacences est souvent préférée, car elle fournit un moyen peu encombrant de représenter les graphes *peu denses* (ceux pour qui $|A|$ est très inférieur à $|S|^2$). La plupart des algorithmes de graphes présentés dans ce livre supposent que le graphe fourni en entrée est représenté sous la forme de listes d'adjacences. Toutefois, la représentation par matrice d'adjacences sera préférable si le graphe est *dense* ($|A|$ est proche de $|S|^2$) ou quand on veut savoir rapidement s'il existe un arc entre deux sommets donnés. Par exemple, deux des algorithmes donnés au chapitre 25 de recherche des plus courts chemins pour tout couple de sommets, supposent que les graphes fournis en entrée sont représentés par des matrices d'adjacences.

La *représentation par listes d'adjacences* d'un graphe $G = (S, A)$ consiste en un tableau Adj de $|S|$ listes, une pour chaque sommet de S . Pour chaque $u \in S$, la liste d'adjacences $Adj[u]$ est une liste des sommets v tels qu'il existe un arc $(u, v) \in A$. Autrement dit, $Adj[u]$ est constituée de tous les sommets adjacents à u dans G . Les sommets de chaque liste d'adjacences sont en général chaînés selon un ordre arbitraire. La figure 22.1(b) montre une représentation par listes d'adjacences du graphe non orienté de la figure 22.1(a). De même, la figure 22.2(b) montre une représentation par listes d'adjacences du graphe orienté de la figure 22.2(a).

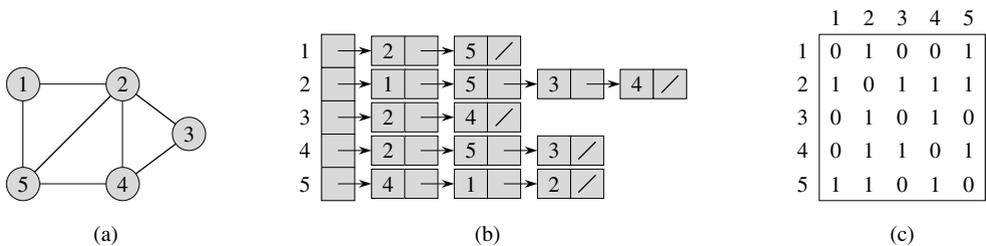


Figure 22.1 Deux représentations d'un graphe non orienté. (a) Un graphe non orienté G possédant cinq sommets et sept arêtes. (b) Une représentation de G par listes d'adjacences. (c) La représentation de G par matrice d'adjacences.

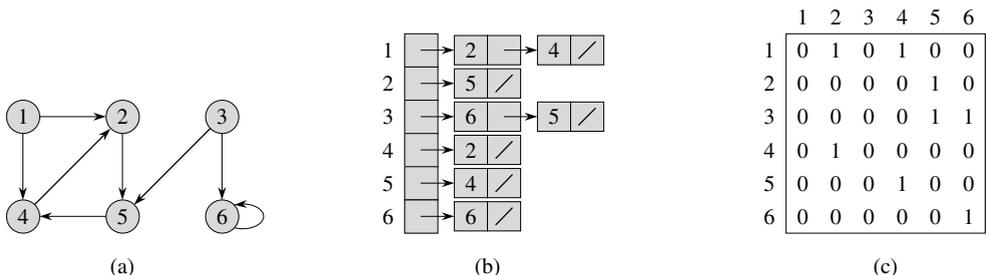


Figure 22.2 Deux représentations d'un graphe orienté. (a) Un graphe orienté G possédant six sommets et huit arcs. (b) Une représentation de G par listes d'adjacences. (c) La représentation de G par matrice d'adjacences.

Si G est un graphe orienté, la somme des longueurs de toutes les listes d'adjacences vaut $|A|$, puisque l'existence d'un arc de la forme (u, v) se traduit par la présence de v dans $Adj[u]$. Si G est un graphe non orienté, la somme des longueurs de toutes les listes d'adjacences vaut $2|A|$, puisque si (u, v) est une arête, u apparaît dans la liste d'adjacences de v , et vice versa. Qu'un graphe soit orienté ou non, la représentation par listes d'adjacences possède la propriété avantageuse de ne demander qu'une quantité de mémoire en $O(\max(S, A)) = O(S + A)$.

Les listes d'adjacences peuvent aisément être adaptées aux **graphes pondérés**, c'est-à-dire aux graphes dont chaque arc possède un **poïds** associé, donné habituellement par une **fonction de pondération** $w : A \rightarrow \mathbf{R}$. Par exemple, soit $G = (S, A)$ un graphe pondéré de fonction de pondération w . Le poids $w(u, v)$ de l'arc $(u, v) \in A$ est tout simplement stocké avec le sommet v dans la liste d'adjacences de u . La représentation par listes d'adjacences est assez robuste, dans le sens où elle peut être modifiée pour supporter de nombreuses variantes sur le graphe.

Un inconvénient potentiel de la représentation par listes d'adjacences est que, pour déterminer si un arc (u, v) donné est présent dans le graphe, il n'existe pas de moyen plus rapide que de rechercher v dans la liste d'adjacences $Adj[u]$. On peut remédier à cet inconvénient en représentant le graphe par une matrice d'adjacences, ce qui se traduira par une utilisation asymptotiquement plus importante de la mémoire.

Pour la **représentation par matrice d'adjacences** d'un graphe $G = (S, A)$, on suppose que les sommets sont numérotés arbitrairement $1, 2, \dots, |S|$. La représentation par matrice d'adjacences d'un graphe G consiste alors en une matrice $|S| \times |S|$, $M = (a_{ij})$ telle que

$$a_{ij} = \begin{cases} 1 & \text{si } (i, j) \in A, \\ 0 & \text{sinon.} \end{cases}$$

Les figures 22.1(c) et 22.2(c) sont les matrices d'adjacences des graphes non orientés et orientés représentés respectivement sur les figures 22.1(a) et 22.2(a). La matrice d'adjacences d'un graphe nécessite une quantité de mémoire en $\Theta(S^2)$, quel que soit le nombre d'arcs du graphe.

On remarque que la matrice d'adjacences de la figure 22.1(c) est symétrique par rapport à sa diagonale principale. On définit la **transposée** d'une matrice $M = (a_{ij})$ comme la matrice ${}^T M = ({}^T a_{ij})$ donnée par ${}^T a_{ij} = a_{ji}$. Comme dans un graphe non orienté, (u, v) et (v, u) représentent la même arête, la matrice d'adjacences M d'un graphe non orienté est sa propre transposée : $M = {}^T M$. Dans certaines applications, il est intéressant de ne conserver que les composantes situées sur et au-dessus de la diagonale de la matrice d'adjacences, ce qui réduit presque de moitié la quantité de mémoire requise pour stocker le graphe.

Comme les listes d'adjacences, les matrices d'adjacences peuvent aussi servir à représenter les graphes pondérés. Par exemple, si $G = (S, A)$ est un graphe pondéré associé à une fonction de pondération w , le poids $w(u, v)$ de l'arc $(u, v) \in A$ est simplement stocké à l'intersection de la ligne u et de la colonne v de la matrice d'adjacences. S'il n'existe aucun arc entre les deux sommets, on peut placer une constante NIL dans la composante correspondante de la matrice, bien que pour de nombreux problèmes, il soit plus pratique d'utiliser une valeur comme 0 ou ∞ .

Quoique la représentation par listes d'adjacences soit asymptotiquement au moins aussi efficace que la représentation par matrice d'adjacences, la simplicité d'une matrice d'adjacences peut rendre son utilisation préférable quand les graphes sont raisonnablement petits. Par ailleurs, si le graphe n'est pas pondéré, cette représentation comporte un avantage supplémentaire. Au lieu d'utiliser un mot mémoire pour chaque composante de matrice, la matrice d'adjacences n'utilise qu'un seul bit.

Exercices

22.1.1 Étant donnée une représentation par listes d'adjacences d'un graphe orienté, combien de temps faut-il pour calculer le degré sortant de tous les sommets ? Et pour calculer les degrés entrants ?

22.1.2 Donner une représentation par listes d'adjacences pour une arborescence binaire complet à 7 sommets. Donner la représentation équivalente par matrice d'adjacences. On suppose que les sommets sont numérotés de 1 à 7 comme dans un tas binaire.

22.1.3 La *transposée* d'un graphe orienté $G = (S, A)$ est le graphe ${}^T G = (S, {}^T A)$, où ${}^T A = \{(v, u) \in S \times S : (u, v) \in A\}$. Autrement dit, ${}^T G$ est obtenu en inversant le sens de tous les arcs de G . Décrire des algorithmes efficaces permettant de calculer ${}^T G$ à partir de G , quand G est représenté par des listes d'adjacences et par une matrice d'adjacences. Analyser le temps d'exécution de vos algorithmes.

22.1.4 Étant donnée une représentation par listes d'adjacences d'un multigraphe $G = (S, A)$, décrire un algorithme en $O(S + A)$ permettant de calculer la représentation par listes d'adjacences du graphe non orienté $G' = (S, A')$ « équivalent », où A' est constitué des arcs de A , mais avec toutes les arcs multiples entre deux sommets remplacées par une arête unique, et toutes les boucles supprimées.

22.1.5 Le *carré* d'un graphe orienté $G = (S, A)$ est le graphe $G^2 = (S, A^2)$ tel que $(u, w) \in A^2$ si et seulement s'il existe un $v \in S$, tel que $(u, v) \in A$ et $(v, w) \in A$. Autrement dit, G^2 possède un arc entre u et w chaque fois que G contient un chemin de longueur deux exactement entre u et w . Décrire des algorithmes efficaces permettant de calculer G^2 à partir de G , quand G est représenté par listes d'adjacences, puis par matrice d'adjacences. Analyser le temps d'exécution de vos algorithmes.

22.1.6 Quand on utilise la représentation par matrice d'adjacences, la plupart des algorithmes de graphes s'exécutent en $\Theta(S^2)$, mais il y a des exceptions. Montrer qu'il est possible de déterminer en $O(S)$ si un graphe orienté contient un **trou noir**, c'est-à-dire un sommet de degré entrant $|S| - 1$ et de degré sortant 0 si on utilise une représentation par matrice d'adjacences.

22.1.7 La **matrice d'incidence** d'un graphe orienté $G = (S, A)$ est une matrice $|S| \times |A|$, $B = (b_{ij})$, telle que

$$b_{ij} = \begin{cases} -1 & \text{si l'arc } j \text{ sort du sommet } i, \\ 1 & \text{si l'arc } j \text{ arrive au sommet } i, \\ 0 & \text{sinon.} \end{cases}$$

Décrire ce que représentent les composantes de la matrice $B^T B$, où ${}^T B$ est la transposée de B .

22.2 PARCOURS EN LARGEUR

Le **parcours en largeur** est l'un des algorithmes de parcours les plus simples, et il est à la base de nombreux algorithmes importants sur les graphes. L'algorithme de Dijkstra pour calculer les plus courts chemins à origine unique (section 3.2) et l'algorithme de Prim pour trouver l'arbre couvrant minimum (section 4.3) utilisent des idées similaires à celles qui sont appliquées lors d'un parcours en largeur.

Étant donné un graphe $G = (S, A)$ et un sommet **origine** s , le parcours en largeur emprunte systématiquement les arcs de G pour « découvrir » tous les sommets accessibles depuis s . Il calcule la distance (plus petit nombre d'arcs) entre s et tous les sommets accessibles. Il construit également une « arborescence de parcours en largeur » de racine s , qui contient tous les sommets accessibles depuis s . Pour tout sommet v accessible depuis s , le chemin reliant s à v dans l'arborescence de parcours en largeur correspond à un « plus court chemin » de s vers v dans G , autrement dit un chemin contenant le plus petit nombre d'arcs. L'algorithme fonctionne aussi bien sur les graphes orientés que sur les graphes non orientés.

L'algorithme de parcours en largeur tient son nom au fait qu'il découvre d'abord tous les sommets situés à une distance k de s avant de découvrir tout sommet situé à la distance $k + 1$.

Pour garder une trace de la progression, le parcours en largeur colorie chaque sommet en blanc, gris, ou noir. Tous les sommets sont blancs au départ, et peuvent devenir gris, puis noirs. Un sommet est **découvert** la première fois qu'il est rencontré au cours de la recherche ; il perd alors sa couleur blanche. Les sommets gris ou noirs ont donc été découverts, et la distinction entre gris et noir permet de garantir que la recherche se poursuit *en largeur d'abord*. Si $(u, v) \in A$ et si le sommet u est noir, alors le sommet v est gris ou noir ; autrement dit, tout sommet adjacent à un sommet noir a déjà été découvert. Les sommets gris peuvent avoir des sommets adjacents blancs ; ils représentent la frontière entre les sommets découverts et les sommets non découverts.

Le parcours en largeur construit une arborescence de parcours en largeur, qui ne contient initialement que sa propre racine, représentant le sommet origine s . À chaque découverte d'un sommet v blanc pendant le balayage de la liste d'adjacences d'un sommet u déjà découvert, le sommet v et l'arc (u, v) sont ajoutés à l'arborescence. On dit que u est le *prédécesseur* ou *parent* de v dans l'arborescence de parcours en largeur. Comme un sommet est découvert au plus une fois, il possède au plus un parent. Les relations ancêtre/descendant dans l'arborescence de parcours en largeur sont définies relativement à la racine s de la manière habituelle : si u se trouve sur un chemin de l'arbre entre la racine s et le sommet v , alors u est un ancêtre de v et v est un descendant de u .

La procédure PL ci-dessous de parcours en largeur suppose que le graphe d'entrée $G = (S, A)$ est représenté par des listes d'adjacences. Pour chaque sommet du graphe, elle maintient à jour plusieurs structures de données supplémentaires. La couleur de chaque sommet $u \in S$ est stockée dans la variable $couleur[u]$, et le parent de u est stocké dans la variable $\pi[u]$. Si u n'a pas de parent (par exemple, si $u = s$ ou si u n'a pas été découvert), alors $\pi[u] = \text{NIL}$. La distance calculée par l'algorithme entre l'origine s et le sommet u est stockée dans $d[u]$. L'algorithme a également recours à une file fifo (premier entré, premier sorti) F (voir section 10.1) pour gérer l'ensemble des sommets gris.

```

PL( $G, s$ )
1  pour chaque sommet  $u \in S[G] - \{s\}$ 
2      faire  $couleur[u] \leftarrow \text{BLANC}$ 
3           $d[u] \leftarrow \infty$ 
4           $\pi[u] \leftarrow \text{NIL}$ 
5   $couleur[s] \leftarrow \text{GRIS}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $F \leftarrow \{s\}$ 
9  tant que  $F \neq \emptyset$ 
10     faire  $u \leftarrow \text{tête}[F]$ 
11         pour chaque  $v \in \text{Adj}[u]$ 
12             faire si  $couleur[v] = \text{BLANC}$ 
13                 alors  $couleur[v] \leftarrow \text{GRIS}$ 
14                      $d[v] \leftarrow d[u] + 1$ 
15                      $\pi[v] \leftarrow u$ 
16                      $\text{ENFILE}(F, v)$ 
17                      $\text{DÉFILE}(F)$ 
18      $couleur[u] \leftarrow \text{NOIR}$ 

```

La figure 22.3 illustre la progression de PL sur un graphe particulier.

La procédure PL fonctionne de la manière suivante. Les lignes 1–4 colorient tous les sommets en blanc, donnent à $d[u]$ la valeur infinie pour chaque sommet u , et

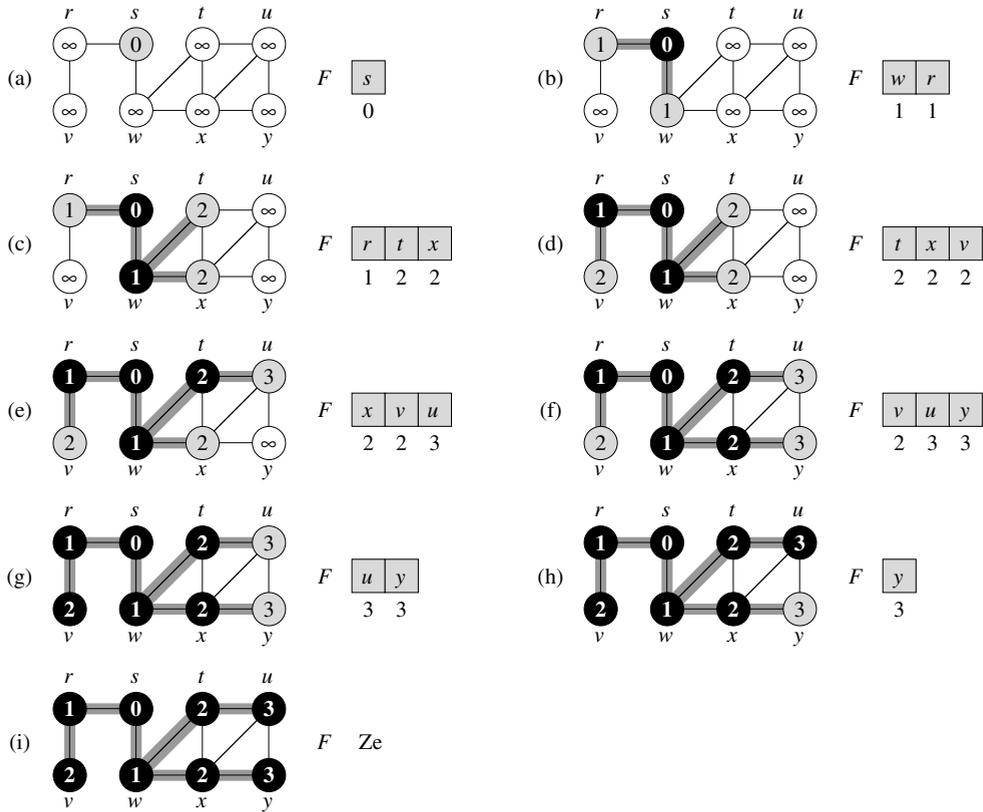


Figure 22.3 L'action de PL sur un graphe non orienté. Les arcs de l'arborescence sont ombrés quand ils sont produits par PL. Les valeurs $d[u]$ sont données à l'intérieur de chaque sommet u . La file F est représentée au début de chaque itération de la boucle **tant que** des lignes 9–18. Les distances à l'origine sont représentées sous chaque sommet de la file.

initialisent à NIL le parent de chaque sommet. La ligne 5 colorie l'origine s en gris, car on convient qu'il est découvert au commencement de la procédure. La ligne 6 initialise $d[s]$ à 0, et la ligne 7 donne au parent de l'origine la valeur NIL. La ligne 8 initialise F en y insérant le seul sommet s ; à partir de là, F contient constamment l'ensemble des sommets gris.

La boucle principale du programme, délimitée par les lignes 10–18, se répète tant qu'il reste des sommets gris, sommets qui ont été découverts mais dont la liste d'adjacences n'a pas été entièrement examinée. Cette boucle **tant que** conserve l'invariant suivant :

Dans le test fait en ligne 10, la file F se compose des sommets gris.

Nous n'utiliserons pas cet invariant pour prouver la validité, mais il est facile de voir qu'il est vérifié avant la première itération et que chaque itération de la boucle conserve l'invariant. Avant la première itération, l'unique sommet gris, qui est en

même temps l'unique sommet dans F , est l'origine s . La ligne 11 détermine le sommet gris u placé en tête de la file F . La boucle **pour** des lignes 12–17 considère chaque sommet v de la liste d'adjacences de u . Si v est blanc, c'est qu'il n'a pas encore été découvert, ce qui est fait par les lignes 14–17. Il est d'abord colorié en gris, et sa distance $d[v]$ prend la valeur $d[u] + 1$. Ensuite, u est enregistré comme étant le parent de v . Enfin, v est placé en queue de la file F . Lorsque tous les sommets de la liste d'adjacences de u ont été examinés, u est supprimé de F et colorié en noir à la ligne 18. L'invariant est conservé parce que, chaque fois qu'un sommet est colorié en gris (ligne 14) il est placé dans la file (ligne 17), et chaque fois qu'un sommet est ôté de la file (ligne 11) il est colorié en noir (ligne 18).

Le résultat du parcours en profondeur peut dépendre de l'ordre dans lequel les voisins d'un sommet donné sont visités en ligne 12 : l'arborescence de parcours en largeur peut varier, mais les distances d calculées par l'algorithme ne changent pas. (Voir exercice 22.2.4.)

a) Analyse

Avant de démontrer toutes les propriétés du parcours en largeur, nous allons, travail relativement aisé, analyser son temps d'exécution sur un graphe $G = (S, A)$. Nous emploierons l'analyse par agrégat, telle que définie à la section 17.1. Après l'initialisation, plus aucun sommet n'est colorié en blanc, et le test de la ligne 13 garantit donc que chaque sommet est enfilé au plus une fois et, partant, défilé au plus une fois. Puisque les opérations d'enfilement et de défilement sont en $O(1)$, le temps total des opérations de file est $O(S)$. Comme la liste d'adjacences de chaque sommet n'est balayée qu'au moment où le sommet est défilé, la liste d'adjacences de chaque sommet est parcourue au plus une fois. La somme des longueurs de toutes les listes d'adjacences étant $\Theta(A)$, le temps total consacré au balayage des listes d'adjacences est $O(A)$. Le coût de l'initialisation est $O(S)$, et le temps d'exécution total de PL est donc $O(S+A)$. Donc, le parcours en largeur s'exécute en un temps qui est linéaire par rapport à la taille de la représentation par listes d'adjacences de G .

b) Plus courts chemins

Au début de cette section, nous avons affirmé qu'un parcours en largeur trouve la distance entre une origine $s \in S$ donnée et chaque sommet accessible depuis s dans un graphe $G = (S, A)$. On définit la **distance de plus court chemin** $\delta(s, v)$ de s à v comme étant le nombre minimal d'arcs d'un chemin reliant le sommet s au sommet v , ou ∞ s'il n'existe aucun chemin de s à v . On dit qu'un chemin de longueur $\delta(s, v)$ de s à v est un **plus court chemin** ⁽¹⁾ de s à v . Avant de montrer que le parcours en largeur

(1) Aux chapitres 24 et 25, nous généraliserons notre étude des plus courts chemins aux graphes pondérés, pour lesquels chaque arc possède un poids à valeur réelle, et où la longueur d'un chemin est la somme des poids des arcs qui le constituent. Les graphes considérés dans le présent chapitre ne sont pas pondérés, c'est-à-dire que tous les arcs ont le même poids unitaire.

calculé effectivement les distances de plus court chemin, nous allons examiner une propriété importante des distances de plus court chemin.

Lemme 22.1 *Soit $G = (S, A)$ un graphe orienté ou non, et soit $s \in S$ un sommet arbitraire. Alors, pour tout arc $(u, v) \in A$,*

$$\delta(s, v) \leq \delta(s, u) + 1 .$$

Démonstration : Si u est accessible depuis s , alors v l'est aussi. Dans ce cas, le plus court chemin de s à v ne peut pas être plus long que le plus court chemin de s à u prolongé par l'arc (u, v) , et l'inégalité est donc valable. Si u ne peut pas être atteint à partir de s , alors $\delta(s, u) = \infty$, et l'inégalité est vérifiée. \square

On souhaite montrer que PL calcule correctement $d[v] = \delta(s, v)$ pour chaque sommet $v \in S$. On commence par montrer que $d[v]$ est un majorant de $\delta(s, v)$.

Lemme 22.2 *Soit $G = (S, A)$ un graphe orienté ou non ; on suppose que PL est exécutée sur G à partir d'une origine donnée $s \in S$. Alors, quand PL se termine, pour tout sommet $v \in S$, la valeur $d[v]$ calculée par PL vérifie l'inégalité $d[v] \geq \delta(s, v)$.*

Démonstration : On utilise une récurrence sur le nombre total de sommets insérés dans la file F . Notre hypothèse de récurrence est que $d[v] \geq \delta(s, v)$ pour tout $v \in S$.

La base de la récurrence se situe juste après que s a été placé dans F à la ligne 9 de PL. L'hypothèse de récurrence est vérifiée ici, car $d[s] = 0 = \delta(s, s)$ et $d[v] = \infty \geq \delta(s, v)$ pour tout $v \in S - \{s\}$.

Pour l'étape de récurrence, on considère un sommet v blanc qui est découvert pendant le balayage des successeurs d'un sommet u . L'hypothèse de récurrence implique que $d[u] \geq \delta(s, u)$. Une fois effectuée l'affectation en ligne 15 et d'après le lemme 22.1, on obtient

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) . \end{aligned}$$

Le sommet v est donc inséré dans la file F , et il ne sera plus jamais inséré puisqu'il est colorié en gris et que la clause **alors** des lignes 14–17 n'est exécutée que pour les sommets blancs. La valeur de $d[v]$ n'est donc plus modifiée, et l'hypothèse de récurrence est maintenue. \square

Pour démontrer que $d[v] = \delta(s, v)$, il faut d'abord caractériser plus précisément le comportement de la file F au cours de l'exécution de PL. Le lemme suivant montre qu'à tout moment, il existe au plus deux valeurs d distinctes dans la file.

Lemme 22.3 *On suppose que pendant l'exécution de PL sur un graphe $G = (S, A)$, la file F contient les sommets $\langle v_1, v_2, \dots, v_r \rangle$, où v_1 est la tête de F et v_r en est la queue. Alors, $d[v_r] \leq d[v_1] + 1$ et $d[v_i] \leq d[v_{i+1}]$ pour $i = 1, 2, \dots, r - 1$.*

Démonstration : La démonstration se fait par récurrence sur le nombre d'opérations de file. Au départ, lorsque la file ne contient que s , le lemme est manifestement vérifié.

Pour l'étape de récurrence, il faut prouver que le lemme est vérifié aussi bien après le défilement qu'après l'enfilement d'un sommet. Si la tête v_1 de la file est défilée, la nouvelle tête devient v_2 . (Si la file devient vide, le lemme est vérifié par défaut.) D'après l'hypothèse de récurrence, $d[v_1] \leq d[v_2]$. Mais alors on a $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, et les autres inégalités restent inchangées. Donc, le lemme est encore valable quand v_2 est en tête.

L'enfilement d'un sommet demande un examen plus attentif du code. Quand on enfile un sommet v en ligne 17 de PL, il devient v_{r+1} . À ce moment-là, on a déjà supprimé le sommet u , dont la liste d'adjacences est en cours de balayage, de la file F , et d'après l'hypothèse de récurrence, la nouvelle tête v_1 vérifie $d[v_1] \geq d[u]$. Donc, $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. D'après l'hypothèse de récurrence, l'on a aussi $d[v_r] \leq d[u] + 1$, et donc $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$, et les autres inégalités restent telles quelles. Par conséquent, le lemme est vérifié quand on enfile v . \square

Le corollaire suivant montre que les valeurs d au moment où les sommets sont enfilés croissent de façon monotone avec le temps.

Corollaire 22.4 *Supposons que les sommets v_i et v_j soient enfilés pendant l'exécution de PL, et que v_i soit enfilé avant v_j . Alors, $d[v_i] \leq d[v_j]$ au moment où v_j est enfilé.*

Démonstration : Immédiate d'après le lemme 22.3 et d'après la propriété selon laquelle chaque sommet reçoit une valeur d finie au plus une fois pendant l'exécution de PL. \square

On peut maintenant prouver que le parcours en largeur trouve correctement les distances de plus court chemin.

Théorème 22.5 (Validité du parcours en largeur) *Soit $G = (S, A)$ un graphe orienté ou non ; supposons que PL est exécuté sur G à partir d'un certain sommet origine $s \in S$. Alors, pendant son exécution, PL découvre chaque sommet $v \in S$ accessible à partir de l'origine s , et à la fin, $d[v] = \delta(s, v)$ pour tout $v \in S$. Par ailleurs, pour tout sommet $v \neq s$ accessible à partir de s , l'un des plus courts chemins de s à v est le plus court chemin de s à $\pi[v]$ complété par l'arc $(\pi[v], v)$.*

Démonstration : Supposons, en raisonnant par l'absurde, qu'un certain sommet reçoive une valeur d qui n'est pas égale à sa distance de plus court chemin. Soit v le sommet ayant le $\delta(s, v)$ minimal qui reçoit une telle valeur d illicite ; visiblement, $v \neq s$. D'après le lemme 22.2, $d[v] \geq \delta(s, v)$, et donc on a $d[v] > \delta(s, v)$. Le sommet v doit être accessible depuis s , car si ce n'est pas le cas, alors $\delta(s, v) = \infty \geq d[v]$. Soit u le sommet qui précède immédiatement v sur un plus court chemin de s à v , de sorte que $\delta(s, v) = \delta(s, u) + 1$. Comme $\delta(s, u) < \delta(s, v)$, et vu la façon dont on a choisi v , on a $d[u] = \delta(s, u)$. En combinant ces propriétés, on obtient

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 . \quad (22.1)$$

Considérons maintenant le moment où PL choisit de défile le sommet u de F en ligne 11. À cet instant, le sommet v est soit blanc, soit gris, soit noir. Nous allons montrer que, dans chacun de ces cas, on arrive à une contradiction de l'inégalité (22.1). Si v est blanc, alors la ligne 15 fait $d[v] = d[u] + 1$, ce qui contredit l'inégalité (22.1). Si v est noir, alors c'est qu'il avait déjà été supprimé de la file et, d'après le corollaire 22.4, on a $d[v] \leq d[u]$, ce qui contredit l'inégalité (22.1). Si v est gris, alors il a été colorié en gris à l'occasion du défilement d'un certain sommet w , qui avait été supprimé de F antérieurement à u et pour lequel $d[v] = d[w] + 1$. Mais, d'après le corollaire 22.4, $d[w] \leq d[u]$, et on a donc $d[v] \leq d[u] + 1$, ce qui contredit l'inégalité (22.1).

On en conclut que $d[v] = \delta(s, v)$ pour tout $v \in S$. Tous les sommets accessibles depuis s doivent être découverts, car s'ils ne l'étaient pas, ils auraient des valeurs d infinies. Pour achever la démonstration du théorème, observons que si $\pi[v] = u$, alors $d[v] = d[u] + 1$. Par conséquent, on peut obtenir un plus court chemin de s à v en prenant un plus court chemin de s à $\pi[v]$ puis en traversant l'arc $(\pi[v], v)$. \square

c) Arborescence de parcours en largeur

La procédure PL construit une arborescence de parcours en largeur pendant qu'elle parcourt le graphe, comme le montre la figure 22.3. L'arborescence est représentée par le champ π de chaque sommet. Plus formellement, pour un graphe $G = (S, A)$ d'origine s , on définit le *sous-graphe prédécesseur* de G par $G_\pi = (S_\pi, A_\pi)$, où

$$S_\pi = \{v \in S : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

et

$$A_\pi = \{(\pi[v], v) \in A : v \in S_\pi - \{s\}\} .$$

Le sous-graphe prédécesseur G_π est une *arborescence de parcours en largeur* si S_π est constitué des sommets accessibles à partir de s et si, pour tout $v \in S_\pi$, il existe un chemin élémentaire unique de s à v dans G_π qui est aussi un plus court chemin de s à v dans G . Une arborescence de parcours en largeur est en fait une arborescence, puisqu'elle est connexe et que $|A_\pi| = |S_\pi| - 1$ (voir théorème B.2). Les arcs de A_π sont appelés *arcs de liaison*.

Après exécution de PL à partir d'une origine s sur un graphe G , le lemme suivant montre que le sous-graphe prédécesseur est une arborescence de parcours en largeur.

Lemme 22.6 *Lorsqu'elle est appliquée à un graphe orienté ou non $G = (S, A)$, la procédure PL construit π de manière que le sous-graphe prédécesseur $G_\pi = (S_\pi, A_\pi)$ soit une arborescence de parcours en largeur.*

Démonstration : La ligne 16 de PL affecte u à $\pi[v]$ si et seulement si $(u, v) \in A$ et $\delta(s, v) < \infty$ (c'est-à-dire, si v est accessible depuis s) et S_π contient donc les sommets de S accessibles depuis s . Comme G_π forme une arborescence, d'après le théorème B2, il contient un chemin unique de s à chaque sommet de S_π . En appliquant par récurrence le théorème 22.5, on conclut que chacun des ces chemins est un plus court chemin. \square

La procédure suivante imprime les sommets d'un plus court chemin reliant s à v , en supposant que PL a déjà été exécutée pour calculer l'arborescence des plus courts chemins.

IMPRIMER-CHEMIN(G, s, v)

```

1  si  $v = s$ 
2    alors imprimer  $s$ 
3    sinon si  $\pi[v] = \text{NIL}$ 
4      alors imprimer " il n'existe aucun chemin de "  $s$  " à "  $v$ 
5      sinon IMPRIMER-CHEMIN( $G, s, \pi[v]$ )
6    imprimer  $v$ 

```

Le temps d'exécution de cette procédure est linéaire par rapport au nombre de sommets du chemin imprimés, puisque chaque appel récursif s'applique à un chemin plus court d'un sommet.

Exercices

22.2.1 Donner les valeurs d et π d'un parcours en largeur sur le graphe orienté de la figure 22.2(a), en prenant pour origine le sommet 3.

22.2.2 Donner les valeurs d et π qui résultent d'un parcours en largeur du graphe non orienté de la figure 22.3, en prenant pour origine le sommet u .

22.2.3 Quel est le temps d'exécution de PL si son graphe d'entrée est représenté par une matrice d'adjacences et que l'algorithme est modifié pour gérer ce type d'entrée ?

22.2.4 Montrer que, dans un parcours en largeur, la valeur $d[u]$ affectée à un sommet u est indépendante de l'ordre dans lequel les sommets apparaissent dans chaque liste d'adjacences. En prenant comme exemple la figure 22.3, montrer que l'arborescence de parcours en largeur calculée par PL peut dépendre de l'ordre au sein des listes d'adjacences.

22.2.5 Donner un exemple de graphe orienté $G = (S, A)$, de sommet origine $s \in S$ et d'ensemble de trois arcs $A_\pi \subseteq A$ tels que, pour chaque sommet $v \in S$, le chemin unique dans A_π de s à v soit un plus court chemin dans G , sans que l'ensemble d'arcs A_π puisse être obtenu en exécutant PL sur G , quel que soit l'ordre des sommets dans chaque liste d'adjacences.

22.2.6 Il existe deux types de catcheurs : les « bons » et les « méchants ». Entre deux catcheurs donnés, il peut ou non y avoir une rivalité. Supposez que l'on ait n catcheurs et une liste de r paires de catcheurs telle qu'il y ait rivalité entre les deux membres de chaque paire. Donner un algorithme à temps $(n + r)$ qui détermine s'il est possible de désigner certains catcheurs comme étant les bons et les autres comme étant les méchants, de telle sorte que chaque rivalité oppose un bon à un méchant. S'il est possible de faire ce genre de classification, votre algorithme doit la calculer.

22.2.7 ★ Le *diamètre* d'une arborescence $T = (S, A)$ est donné par

$$\max_{u, v \in S} \delta(u, v);$$

autrement dit, le diamètre est la plus grande de toutes les distances de plus court chemin dans l'arborescence. Donner un algorithme efficace permettant de calculer le diamètre d'une arborescence, et analyser son temps d'exécution.

22.2.8 Soit $G = (S, A)$ un graphe connexe, non orienté. Donner un algorithme en $O(S + A)$ permettant de calculer une chaîne de G qui traverse chaque arête de A exactement une fois dans chaque direction. Décrire une manière de trouver son chemin dans un labyrinthe quand on dispose d'une grande quantité de pièces de monnaie.

22.3 PARCOURS EN PROFONDEUR

La stratégie suivie par un parcours en profondeur est, comme son nom l'indique, de descendre plus « profondément » dans le graphe chaque fois que c'est possible. Lors d'un parcours en profondeur, les arcs sont explorés à partir du sommet v découvert le plus récemment et dont on n'a pas encore exploré tous les arcs qui en partent. Lorsque tous les arcs de v ont été explorés, l'algorithme « revient en arrière » pour explorer les arcs qui partent du sommet à partir duquel v a été découvert. Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir du sommet origine initial aient été découverts. S'il reste des sommets non découverts, on en choisit un qui servira de nouvelle origine, et le parcours reprend à partir de cette origine. Le processus complet est répété jusqu'à ce que tous les sommets aient été découverts.

Comme dans le parcours en largeur, chaque fois qu'un sommet v est découvert pendant le balayage d'une liste d'adjacences d'un sommet u , le parcours en profondeur enregistre cet événement en donnant la valeur u à $\pi[v]$, parent de v . Contrairement au parcours en largeur, pour lequel le sous-graphe prédécesseur forme une arborescence, le sous-graphe prédécesseur obtenu par un parcours en profondeur peut être composé de plusieurs arborescences, car le parcours peut être répété à partir de plusieurs origines. ⁽²⁾ Le **sous-graphe prédécesseur** d'un parcours en profondeur est donc défini un peu différemment de celui d'un parcours en largeur : on pose $G_\pi = (S, A_\pi)$, où

$$A_\pi = \{(\pi[v], v) : v \in S \text{ et } \pi[v] \neq \text{NIL}\} .$$

Le sous-graphe prédécesseur d'un parcours en profondeur forme une **forêt de parcours en profondeur** composée de plusieurs **arborescences de parcours en profondeur**. Les arcs de A_π sont des **arcs de liaison**.

Comme dans le parcours en largeur, les sommets sont coloriés pendant le parcours pour indiquer leur état. Chaque sommet est initialement blanc, puis gris quand il est **découvert** pendant le parcours, et enfin noir en fin de traitement, c'est-à-dire quand sa liste d'adjacences a été complètement examinée. Cette technique assure que chaque sommet appartient à une arborescence de parcours en profondeur et un seul, de sorte que ces arborescences sont disjointes.

(2) Il peut paraître arbitraire que le parcours en largeur soit limité à une seule origine alors que le parcours en profondeur peut partir de plusieurs origines. Théoriquement, rien n'empêche de faire un parcours en largeur à partir de plusieurs sources, ni de limiter un parcours en profondeur à une origine unique, mais notre exposé reflète ce qui se fait généralement. Le parcours en largeur s'utilise le plus souvent pour trouver des distances de plus court chemin (et le sous-graphe prédécesseur associé) à partir d'une origine donnée. Le parcours en profondeur est souvent une sous-routine insérée dans un autre algorithme, comme nous le verrons plus loin.

En plus de créer une forêt de parcours en profondeur, le parcours en profondeur *date* chaque sommet. Chaque sommet v porte deux dates : la première, $d[v]$, marque le moment où v a été découvert pour la première fois (et colorié en gris), et la seconde, $f[v]$, enregistre le moment où le parcours a fini d'examiner la liste d'adjacences de v (et le colorie en noir). Ces dates sont utilisées dans de nombreux algorithmes de graphes et sont utiles pour analyser le comportement du parcours en profondeur.

La procédure PP ci-après enregistre le moment où elle découvre le sommet u dans la variable $d[u]$, et le moment où elle termine le traitement du sommet u dans la variable $f[u]$. Ces dates sont des entiers compris entre 1 et $2|S|$, puisque découverte et fin de traitement se produisent une fois et une seule pour chacun des $|S|$ sommets. Pour tout sommet u ,

$$d[u] < f[u]. \quad (22.2)$$

Le sommet u est BLANC avant l'instant $d[u]$, GRIS entre $d[u]$ et $f[u]$, et NOIR après.

Le pseudo code suivant correspond à l'algorithme de base de parcours en profondeur. Le graphe d'entrée G peut être orienté ou non. La variable *date* est une variable globale qui sert à la datation.

PP(G)

```

1  pour chaque sommet  $u \in S[G]$ 
2    faire  $couleur[u] \leftarrow$  BLANC
3     $\pi[u] \leftarrow$  NIL
4   $date \leftarrow 0$ 
5  pour chaque sommet  $u \in S[G]$ 
6    faire si  $couleur[u] =$  BLANC
7      alors VISITER-PP( $u$ )

```

VISITER-PP(u)

```

1   $couleur[u] \leftarrow$  GRIS  $\triangleright$  sommet blanc  $u$  vient d'être découvert.
2   $date \leftarrow date + 1$ 
3   $d[u] \leftarrow date$ 
4  pour chaque  $v \in Adj[u] \triangleright$  Exploration de l'arc  $(u, v)$ .
5    faire si  $couleur[v] =$  BLANC
6      alors  $\pi[v] \leftarrow u$ 
7      VISITER-PP( $v$ )
8   $couleur[u] \leftarrow$  NOIR  $\triangleright$  noircir  $u$ , car on en a fini avec lui.
9   $f[u] \leftarrow date \leftarrow date + 1$ 

```

La figure 22.4 illustre la progression de PP sur le graphe de la figure 22.2.

La procédure PP fonctionne de la manière suivante. Les lignes 1–3 colorient tous les sommets en blanc et initialisent leurs champs π à NIL. La ligne 4 initialise le compteur de dates. Les lignes 5–7 testent chaque sommet de S l'un après l'autre

et, quand un sommet blanc est trouvé, le visitent grâce à VISITER-PP. À chaque appel VISITER-PP(u) à la ligne 7, le sommet u devient la racine d'une nouvelle arborescence de la forêt de parcours en profondeur. Quand PP se termine, chaque sommet u s'est vu affecter une *date de découverte* $d[u]$ et une *date de fin de traitement* $f[u]$.

Pour chaque appel VISITER-PP(u), le sommet u est blanc initialement. La ligne 1 colorie u en gris, la ligne 2 incrémente la variable globale *date* et la ligne 3 enregistre la nouvelle valeur de *date* pour en faire la date de découverte $d[u]$. Les lignes 4–7 examinent chaque sommet v adjacent à u et visitent récursivement v s'il est blanc. Comme chaque sommet $v \in Adj[u]$ est considéré à la ligne 4, on dit que l'arc (u, v) est *exploré* par le parcours en profondeur. Enfin, après l'exploration de tous les arcs partant de u , les lignes 8–9 peignent u en noir et enregistrent la date de fin de traitement dans $f[u]$.

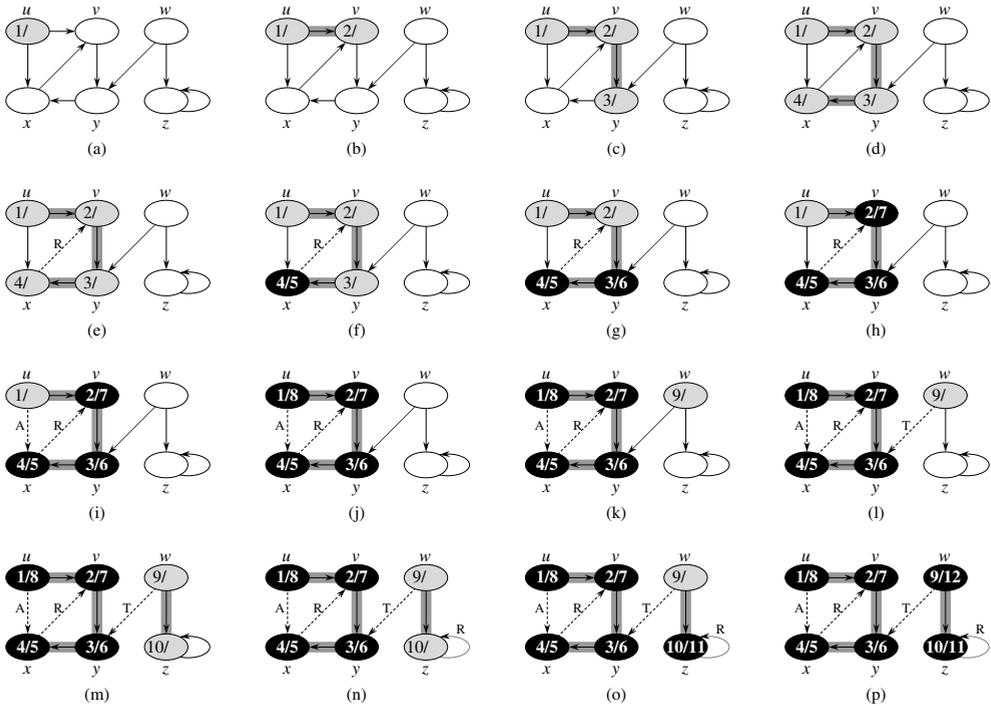


Figure 22.4 La progression de l'algorithme de recherche en profondeur PP sur un graphe orienté. Au fur et à mesure que les arcs sont explorés, ils sont représentés sur fond ombré (si ce sont des arcs de liaison) ou tracés en pointillés (sinon). Les arcs ne faisant pas partie des arborescences (arcs qui ne sont pas des arcs de liaison) sont étiquetés R, T ou A selon que ce sont des arcs arrières, transverses ou avant. Les sommets sont étiquetés par leur dates de découverte et de fin de traitement.

Notez que le résultat du parcours en profondeur peut dépendre de l'ordre dans lequel les sommets sont examinés en ligne 5 de PP, et de l'ordre dans lequel les voisins d'un sommet sont visités en ligne 4 de VISITER-PP. Ces ordres de visite différents ne posent généralement pas problème dans la pratique, dans la mesure où *chaque* résultat d'une recherche en profondeur fournit généralement des sorties équivalentes.

Quel est le temps d'exécution de PP ? Les boucles des lignes 1–3 et 5–7 requièrent un temps $\Theta(S)$, sans compter le temps d'exécution des appels VISITER-PP. La procédure VISITER-PP est appelée exactement une fois pour chaque sommet $v \in S$, puisqu'elle n'est invoquée que sur les sommets blancs, et qu'elle commence par les peindre en gris. Pendant l'exécution de VISITER-PP(v), la boucle des lignes 4–7 est exécutée $|Adj[v]|$ fois. Comme

$$\sum_{v \in S} |Adj[v]| = \Theta(A),$$

le coût total d'exécution des lignes 4–7 de VISITER-PP est $\Theta(A)$. Le temps d'exécution de PP est donc $\Theta(S + A)$.

a) Propriétés du parcours en profondeur

Le parcours en profondeur révèle beaucoup d'informations sur la structure d'un graphe. La propriété la plus fondamentale du parcours en profondeur est sans doute que le sous-graphe prédécesseur G_π forme en fait une forêt, puisque la structure des arborescences de parcours en profondeur reflète exactement la structure des appels récursifs à VISITER-PP. Autrement dit, $u = \pi[v]$ si et seulement si VISITER-PP(v) a été appelé pendant le parcours de la liste d'adjacences de u . En outre, le sommet v est un descendant du sommet u dans la forêt de parcours en profondeur si et seulement si v est découvert pendant que u est gris.

Une autre propriété importante du parcours en profondeur tient au fait que les dates de découverte et de fin de traitement ont une **structure parenthésée**. Si l'on représente la découverte d'un sommet u par une parenthèse gauche « $(u$ » et la fin de son traitement par une parenthèse droite « $u)$ », alors la succession des découvertes et des fins de traitement crée une expression bien formée, au sens où les parenthèses sont correctement imbriquées. Par exemple, le parcours en profondeur de la figure 22.5(a) correspond au parenthésage montré à la figure 22.5(b). Une autre manière d'énoncer la condition de structure parenthésée est donnée par le théorème suivant.

Théorème 22.7 (Théorème des parenthèses) *Dans un parcours en profondeur d'un graphe $G = (S, A)$ (orienté ou non), pour deux sommets quelconques u et v , une et une seule des trois conditions suivantes est vérifiée :*

- les intervalles $[d[u], f[u]]$ et $[d[v], f[v]]$ sont complètement disjoints, et ni u ni v n'est un descendant de l'autre dans la forêt de parcours en profondeur,

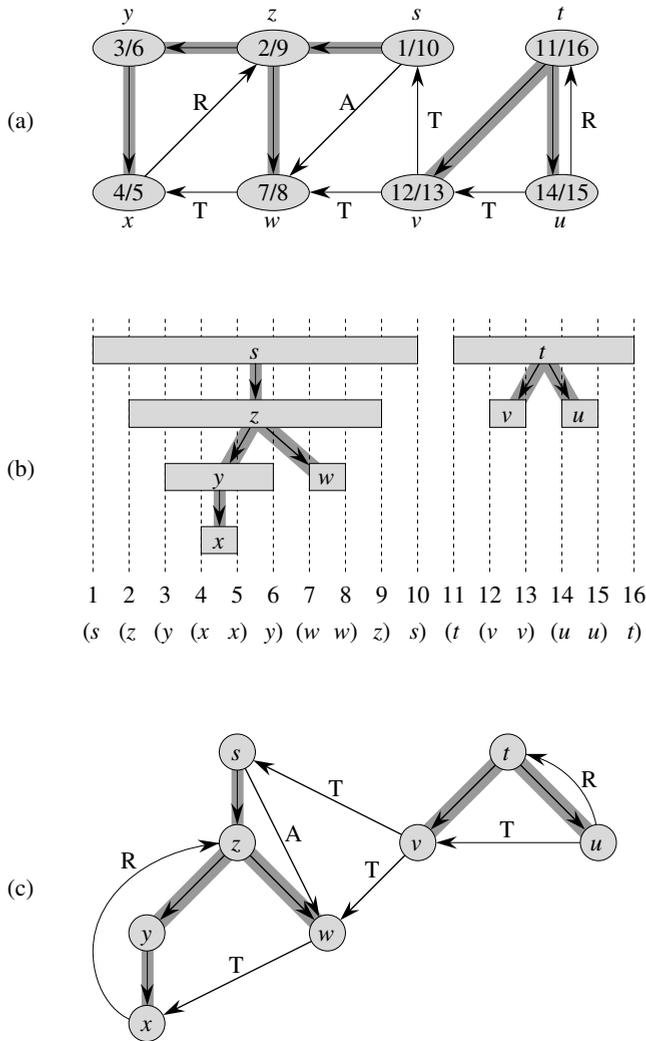


Figure 22.5 Propriétés du parcours en profondeur. (a) Le résultat d'un parcours en profondeur sur un graphe orienté. Les sommets sont datés et les types d'arc sont indiqués comme dans la figure 22.4. (b) Les intervalles des dates de découverte et de fin de traitement de chaque sommet correspondent au parenthésage montré. Chaque rectangle représente l'intervalle de temps entre la date de découverte et celle de fin de traitement du sommet correspondant. Les arcs de liaison sont représentés. Si deux intervalles se recoupent, alors l'un est inclus dans l'autre et le sommet correspondant au plus petit intervalle est un descendant du sommet correspondant à l'intervalle plus grand. (c) Le graphe de la partie (a) redessiné avec tous les arcs de liaison et tous les arcs avant qui descendent dans un arbre en profondeur, et avec tous les arcs arrière qui remontent d'un descendant vers un ancêtre.

- l'intervalle $[d[u], f[u]]$ est entièrement inclus dans l'intervalle $[d[v], f[v]]$, et u est un descendant de v dans une arborescence de parcours en profondeur; ou
- l'intervalle $[d[v], f[v]]$ est entièrement inclus dans l'intervalle $[d[u], f[u]]$, et v est un descendant de u dans une arborescence de parcours en profondeur.

Démonstration : On commence par le cas $d[u] < d[v]$. Il faut considérer deux sous-cas, selon que $d[v] < f[u]$ ou non. Si $d[v] < f[u]$, c'est que v a été découvert pendant que u était encore gris. Cela implique que v est un descendant de u . Par ailleurs, puisque la découverte de v est plus récente que celle de u , tous les arcs qui en partent sont explorés, et le traitement de v se termine avant que le parcours ne revienne à u et finisse de le traiter. Dans ce cas, l'intervalle $[d[v], f[v]]$ est entièrement contenu dans l'intervalle $[d[u], f[u]]$. Dans l'autre sous-cas, $f[u] < d[v]$, et l'inégalité (22.2) implique que les intervalles $[d[u], f[u]]$ et $[d[v], f[v]]$ sont disjoints. Comme les intervalles sont disjoints, aucun des deux sommets n'a été découvert pendant que l'autre était gris, et donc aucun sommet n'est un descendant de l'autre.

Pour le cas où $d[v] < d[u]$, il suffit d'inverser les rôles de u et v dans la démonstration précédente. \square

Corollaire 22.8 (Imbrications des intervalles des descendants) *Le sommet v est un descendant propre du sommet u dans la forêt de parcours en profondeur d'un graphe G (orienté ou non) si et seulement si $d[u] < d[v] < f[v] < f[u]$.*

Démonstration : Immédiate d'après le théorème 22.7. \square

Le théorème suivant donne une autre caractérisation importante des descendants d'un sommet dans une forêt de parcours en profondeur.

Théorème 22.9 (Théorème du chemin blanc) *Dans une forêt de parcours en profondeur d'un graphe $G = (S, A)$ (orienté ou non), un sommet v est un descendant d'un sommet u si et seulement si, au moment $d[u]$ où le parcours découvre u , il existe un chemin de u à v composé uniquement de sommets blancs.*

Démonstration : \Rightarrow : Supposons que v soit un descendant de u . Soit w un sommet quelconque sur le chemin de u à v dans l'arborescence de parcours en profondeur, de sorte que w est un descendant de u . D'après le corollaire 22.8, $d[u] < d[w]$, et donc w est blanc à l'instant $d[u]$.

\Leftarrow : Supposons qu'à l'instant $d[u]$, il existe un chemin constitué de sommets blancs de u à v , mais que v ne devienne pas un descendant de u dans l'arborescence de parcours en profondeur. On peut, sans perte de généralité, supposer que tous les autres sommets du chemin deviennent des descendants de u . (Si ce n'est pas le cas, on choisit pour v le sommet le plus proche de u , sur le chemin, qui ne devient pas un descendant de u). Soit w le parent de v dans le chemin, de sorte que w est un descendant de u (w et u peuvent être un seul et même sommet) ; d'après le corollaire 22.8, $f[w] \leq f[u]$. Notez que v doit être découvert après u , mais avant que le traitement de w soit terminé. Donc, $d[u] < d[v] < f[w] \leq f[u]$. Le théorème 22.7 implique alors que l'intervalle $[d[v], f[v]]$ est entièrement inclus dans l'intervalle $[d[u], f[u]]$. D'après le corollaire 22.8, v est finalement un descendant de u . \square

b) Classification des arcs

Une autre propriété intéressante du parcours en profondeur est que le parcours peut servir à classer les arcs du graphe d'entrée $G = (S, A)$. Cette classification des arcs peut servir à glaner d'importantes informations concernant le graphe. Par exemple, dans la section suivante, nous verrons qu'un graphe orienté est acyclique si et seulement si un parcours en profondeur n'engendre aucun arc « arrière » (lemme 22.11).

Il est possible de définir quatre types d'arcs en fonction de la forêt de parcours en profondeur G_π obtenue par un parcours en profondeur sur G .

- 1) Les **arcs de liaison** sont les arcs de la forêt de parcours en profondeur G_π . L'arc (u, v) est un arc de liaison si v a été découvert la première fois pendant le parcours de l'arc (u, v) .
- 2) Les **arcs arrières** sont les arcs (u, v) reliant un sommet u à un ancêtre v dans une arborescence de parcours en profondeur. Les boucles (graphes orientés uniquement) sont considérées comme des arcs arrières.
- 3) Les **arcs avants** sont les arcs (u, v) qui ne sont pas des arcs de liaison, et qui relient un sommet u à un descendant v dans une arborescence de parcours en profondeur.
- 4) Les **arcs transverses** sont tous les autres arcs. Ils peuvent relier deux sommets d'une même arbre de parcours en profondeur, du moment que l'un des sommets n'est pas un ancêtre de l'autre ; ils peuvent aussi relier deux sommets appartenant à des arborescences de parcours en profondeur différentes.

Sur les figures 22.4 et 22.5, les arcs portent des étiquettes indiquant leur type. La figure 22.5(c) montre également comment le graphe de la figure 22.5(a) peut être redessiné de manière que tous les arcs de liaison et tous les arcs avants soient orientés vers le bas dans une arborescence de parcours en profondeur, et que tous les arcs arrières pointent vers le haut. Tout graphe peut être redessiné de cette façon.

L'algorithme PP peut être modifié pour classer les arcs à mesure qu'il les rencontre. Le principe est que chaque arc (u, v) peut être classé en fonction de la couleur du sommet v atteint lorsque l'arc est exploré pour la première fois (excepté que les arcs avants et les arcs transverses ne sont pas différenciés) :

- 1) BLANC indique un arc de liaison,
- 2) GRIS indique un arc arrière, et
- 3) NOIR indique un arc avant ou transverse.

Le premier cas se déduit immédiatement de la spécification de l'algorithme. Pour le deuxième cas, on observe que les sommets gris forment toujours une chaîne linéaire de descendants correspondant à la pile courante des invocations de VISITER-PP ; le nombre de sommets gris vaut un de plus que la profondeur, dans la forêt de parcours en profondeur, du sommet le plus récemment découvert. L'exploration part toujours du sommet gris le plus profond, et donc un arc qui atteint un autre sommet gris atteint un ancêtre. Le troisième cas regroupe les autres possibilités : on peut montrer

qu'un arc (u, v) de ce type est un arc avant si $d[u] < d[v]$ et est un arc transverse si $d[u] > d[v]$. (Voir exercice 22.3.4.)

Dans un graphe non orienté, la classification peut engendrer une ambiguïté, car (u, v) et (v, u) forment en réalité une seule et même arête. Dans ce cas, on convient que l'arc est du *premier* type applicable dans la liste de classification. Autrement dit (voir exercice 22.3.5), l'arc est classé selon que (u, v) ou (v, u) est rencontré en premier pendant l'exécution de l'algorithme.

Nous allons maintenant montrer qu'aucun arc avant ou transverse n'apparaît pendant le parcours en profondeur d'un graphe non orienté.

Théorème 22.10 *Dans un parcours en profondeur d'un graphe non orienté G , chaque arête de G est soit un arc de liaison, soit un arc arrière.*

Démonstration : Soit (u, v) une arête arbitraire de G ; on suppose, sans perte de généralité, que $d[u] < d[v]$. Alors, v doit être découvert et son traitement terminé avant la fin du traitement de u (pendant que u est gris), puisque v se trouve dans la liste d'adjacences de u . Si l'arête (u, v) est d'abord explorée dans le sens u vers v , alors v n'a pas été découvert (blanc) jusqu'à ce moment, car sinon on aurait déjà exploré cette arête dans la direction de v vers u . Donc, (u, v) devient un arc de liaison. Si (u, v) est d'abord exploré dans la direction de v vers u , alors (u, v) est un arc arrière, car u est encore gris au moment où l'arête est explorée pour la première fois. \square

Nous verrons de nombreuses applications de ces théorèmes dans les sections suivantes.

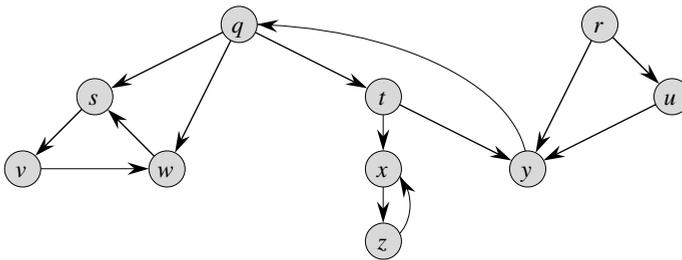


Figure 22.6 Un graphe orienté servant de support aux exercices 22.3.2 et 22.5.2.

Exercices

22.3.1 Dessiner un tableau 3×3 en étiquetant les lignes et les colonnes avec BLANC, GRIS et NOIR. Dans chaque cellule (i, j) , indiquer s'il peut exister, pendant le parcours en profondeur d'un graphe orienté, un arc reliant un sommet de couleur i à un sommet de couleur j . Pour chaque arc possible, indiquer de quel(s) type(s) il peut être. Dessiner un second tableau pour le parcours en profondeur d'un graphe non orienté.

22.3.2 Donner le déroulement du parcours en profondeur sur le graphe de la figure 22.6. On supposera que la boucle **pour** des lignes 5–7 de la procédure PP considère les sommets dans l'ordre alphabétique, et que chaque liste d'adjacences est ordonnée alphabétiquement. Donner les dates de découverte et de fin de traitement de chaque sommet, ainsi que la classification de chaque arc.

22.3.3 Mettre en évidence la structure parenthésée du parcours en profondeur illustré à la figure 22.4.

22.3.4 Montrer que l'arc (u, v) est

- un arc de liaison ou un arc avant si et seulement si $d[u] < d[v] < f[v] < f[u]$,
- un arc arrière si et seulement si $d[v] < d[u] < f[u] < f[v]$, et
- un arc transverse si et seulement si $d[v] < f[v] < d[u] < f[u]$.

22.3.5 Montrer que, dans un graphe non orienté, la classification d'une arête (u, v) comme arc de liaison ou arc arrière selon que le parcours en profondeur rencontre en premier (u, v) ou (v, u) équivaut à une classification selon la priorité des types du schéma de classification.

22.3.6 Réécrire PP, en utilisant une pile pour supprimer la récursivité.

22.3.7 Donner un contre-exemple de la conjecture selon laquelle s'il existe un chemin entre u et v dans un graphe orienté G , et si $d[u] < d[v]$ lors d'un parcours en profondeur de G , alors v est un descendant de u dans la forêt de parcours en profondeur obtenue.

22.3.8 Donner un contre-exemple de la conjecture selon laquelle s'il existe un chemin entre u et v dans un graphe orienté G , alors tout parcours en profondeur donne forcément $d[v] \leq f[u]$.

22.3.9 Modifier le pseudo code du parcours en profondeur de manière qu'il imprime tous les arcs du graphe orienté G avec leur type. Donner les modifications éventuelles qu'il faudrait effectuer si G était non orienté.

22.3.10 Expliquer comment un sommet u d'un graphe orienté peut se retrouver dans une arborescence de parcours en profondeur contenant seulement u , même si u a à la fois des arcs entrants et des arcs sortants dans G .

22.3.11 Montrer qu'un parcours en profondeur d'un graphe non orienté G peut servir à identifier les composantes connexes de G , et que la forêt de parcours en profondeur contient autant d'arborescences que G possède de composantes connexes. Plus précisément, montrer comment modifier le parcours en profondeur pour que chaque sommet v soit affecté d'une étiquette entière $cc[v]$ comprise entre 1 et k , où k est le nombre de composantes connexes de G , telle que $cc[u] = cc[v]$ si et seulement si u et v appartiennent à la même composante connexe.

22.3.12 * Un graphe orienté $G = (S, A)$ est **simplement connexe** si $u \rightsquigarrow v$ implique qu'il existe au plus un chemin simple de u à v pour tous les sommets $u, v \in S$. Donner un algorithme efficace capable de déterminer si un graphe (orienté ou non) est simplement connexe.

22.4 TRI TOPOLOGIQUE

Cette section montre comment utiliser le parcours en profondeur pour effectuer un tri topologique d'un graphe orienté sans circuit. Le *tri topologique* d'un graphe orienté sans circuit $G = (S, A)$ consiste à ordonner linéairement tous ses sommets de sorte que, si G contient un arc (u, v) , u apparaisse avant v dans le tri. (Si le graphe n'est pas sans circuit, aucun ordre linéaire n'est possible.) Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale de manière que tous les arcs soient orientés de gauche à droite. Le tri topologique diffère donc du « tri » habituel étudié dans la partie 2.

Les graphes orientés sans circuit sont utilisés dans de nombreuses applications pour représenter des précédences entre événements. La figure 22.7 donne l'exemple du savant Cosinus qui s'habille le matin. Le professeur doit enfiler certains vêtements avant d'autres (par exemple les chaussettes avant les chaussures). D'autres peuvent être mis dans n'importe quel ordre (par exemple, les chaussettes et le pantalon). Un arc (u, v) du graphe orienté sans circuit de la figure 22.7(a) indique que le vêtement u doit être enfilé avant le vêtement v . Le tri topologique de ce graphe orienté sans circuit donne donc un ordre permettant de s'habiller correctement. La figure 22.7(b) montre le graphe orienté sans circuit trié topologiquement comme une suite de sommets sur une ligne horizontale de telle façon que tous les arcs soient orientés de gauche à droite.

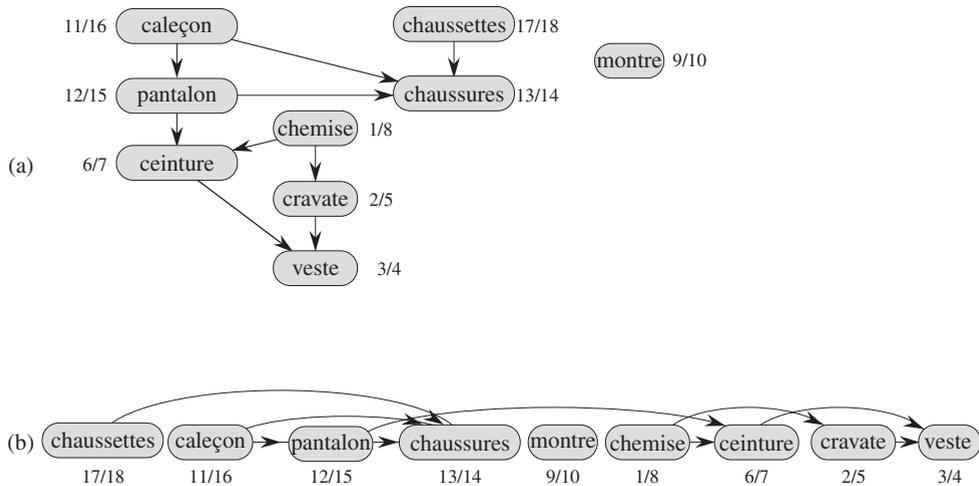


Figure 22.7 (a) Le savant Cosinus trie topologiquement ses vêtements quand il s'habille. Chaque arc (u, v) signifie que le vêtement u doit être enfilé avant le vêtement v . Les dates de découverte et de fin de traitement résultant d'un parcours en profondeur sont données à côté de chaque sommet. (b) Le même graphe trié topologiquement. Ses sommets sont ordonnés de gauche à droite par ordre décroissant des dates de fin de traitement. Notez que tous les arcs sont orientés de gauche à droite.

L'algorithme simple suivant permet d'effectuer le tri topologique d'un graphe orienté sans circuit.

TRI-TOPOLOGIQUE(G)

- 1 appeler PP(G) pour calculer les dates de fin de traitement $f[v]$ pour chaque sommet v
- 2 chaque fois que le traitement d'un sommet se termine, insérer le sommet début d'une liste chaînée
- 3 **retourner** la liste chaînée des sommets

La figure 22.7(b) montre les sommets triés topologiquement selon l'ordre inverse des dates de fin de traitement.

On peut effectuer un tri topologique en $\Theta(S + A)$, car le parcours en profondeur prend $\Theta(S + A)$ et l'insertion de chacun des $|S|$ sommets au début de la liste chaînée nécessite $O(1)$.

On démontre la validité de cet algorithme à l'aide du lemme fondamental suivant, qui caractérise les graphes orientés sans circuit.

Lemme 22.11 *Un graphe orienté G est sans circuit si et seulement si un parcours en profondeur de G ne génère aucun arc arrière.*

Démonstration : \Rightarrow : On suppose qu'il existe un arc arrière (u, v) . Alors, le sommet v est un ancêtre du sommet u dans la forêt de parcours en profondeur. Il existe donc un chemin de v à u dans G , et l'arc arrière (u, v) complète un circuit.

\Leftarrow : Supposons que G contienne un circuit c . On va montrer qu'un parcours en profondeur de G génère un arc arrière. Soit v le premier sommet découvert dans c , et soit (u, v) l'arc précédent dans c . A l'instant $d[v]$, les sommets de c forment un chemin entre v et u composé de sommets blancs. D'après le théorème du chemin blanc, le sommet u devient un descendant de v dans la forêt de parcours en profondeur. (u, v) est donc un arc arrière. \square

Théorème 22.12 TRI-TOPOLOGIQUE(G) effectue le tri topologique d'un graphe orienté sans circuit G .

Démonstration : Supposons que PP soit exécutée sur un graphe orienté sans circuit $G = (S, A)$ donné pour déterminer les dates de fin de traitement de ses sommets. Il suffit de montrer que, pour toute paire de sommets distincts $u, v \in S$, s'il existe un arc dans G menant de u à v , alors $f[v] < f[u]$. On considère un arc (u, v) quelconque exploré par PP(G). Lorsque cet arc est exploré, v ne peut pas être gris, car alors v serait un ancêtre de u , et (u, v) serait un arc arrière, ce qui contredit le lemme 22.11. v est donc soit blanc, soit noir. Si v est blanc, il devient un descendant de u , et donc $f[v] < f[u]$. Si v est noir, c'est que son traitement est achevé, et donc que $f[v]$ a déjà été initialisé. Comme on est encore en train d'explorer à partir de u , il reste encore à dater $f[u]$, et une fois que ce sera fait, on aura derechef $f[v] < f[u]$. Donc, pour un arc (u, v) quelconque du graphe orienté sans circuit, on a $f[v] < f[u]$, ce qui démontre le théorème. \square

Exercices

22.4.1 Donner l'ordre dans lequel TRI-TOPOLOGIQUE trie les sommets quand on l'exécute sur le graphe orienté sans circuit de la figure 22.8, en gardant l'hypothèse de l'exercice 22.3.2.

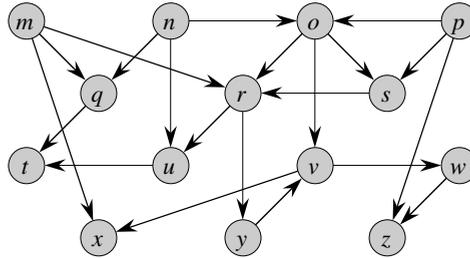


Figure 22.8 Un graphe orienté sans circuit à trier topologiquement.

22.4.2 Donner un algorithme à temps linéaire qui prend en entrée un graphe orienté sans circuit $G = (S, A)$ et deux sommets s et t , puis qui retourne le nombre de chemins allant de s à t dans G . Par exemple, dans le graphe orienté sans circuit de la figure 22.8, il existe exactement quatre chemins permettant de relier le sommet p au sommet v : pov , $poryv$, $posryv$ et $psryv$. (Votre algorithme n'est pas obligé d'énumérer les chemins. Il suffit de les compter.)

22.4.3 Donner un algorithme qui détermine si un graphe non orienté $G = (S, A)$ donné contient ou non un cycle. L'algorithme devra s'exécuter en $O(S)$, indépendamment de $|A|$.

22.4.4 Démontrer ou contredire l'affirmation suivante : si un graphe orienté G contient des circuits, alors TRI-TOPOLOGIQUE(G) fournit un ordre des sommets qui minimise le nombre de « mauvais » arcs qui ne sont pas cohérents avec l'ordre obtenu.

22.4.5 Un autre moyen d'effectuer le tri topologique d'un graphe orienté sans circuit $G = (S, A)$ consiste à faire, de manière itérative, les opérations suivantes : trouver un sommet de degré entrant 0, l'imprimer, puis le supprimer du graphe ainsi que tous les arcs qui en partent. Expliquer comment implémenter cette idée par un algorithme à temps $O(S + A)$. Qu'advient-il de cet algorithme si G contient des circuits ?

22.5 COMPOSANTES FORTEMENT CONNEXES

Nous allons à présent considérer une application classique du parcours en profondeur : la décomposition d'un graphe orienté en ses composantes fortement connexes. Cette section montre comment effectuer cette décomposition à l'aide de deux parcours en profondeur. Beaucoup d'algorithmes de graphes orientés commencent par cette décomposition ; cette approche permet souvent de diviser le problème initial

en sous-problèmes, un par composante fortement connexe. La combinaison des solutions des sous-problèmes se fait en suivant la structure des connexions entre les composantes.

Depuis l'annexe B, nous savons qu'une composante fortement connexe d'un graphe orienté $G = (S, A)$ est un ensemble maximal de sommets $R \subseteq S$ tel que, pour chaque paire de sommets u et v de R , on ait à la fois $u \rightsquigarrow v$ et $v \rightsquigarrow u$; autrement dit, les sommets u et v sont mutuellement accessibles. La figure 22.9 en donne une illustration.

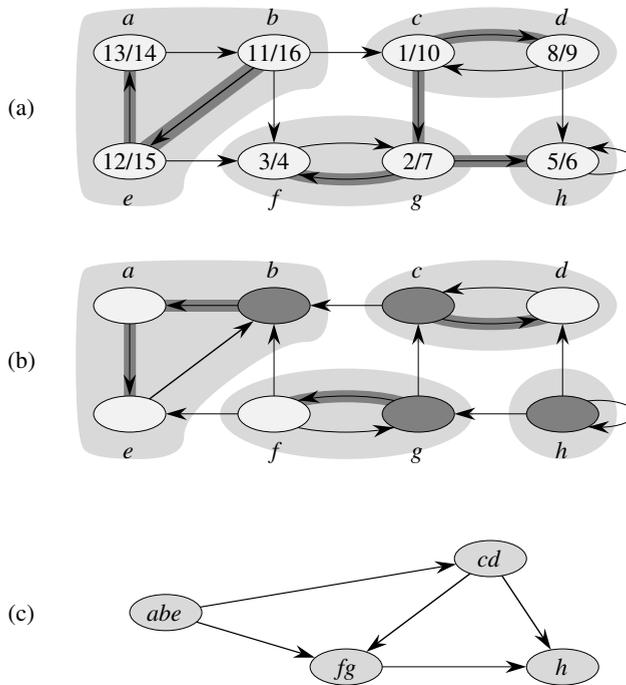


Figure 22.9 (a) Un graphe orienté G . Les composantes fortement connexes de G sont matérialisées par des régions en gris. Chaque sommet est étiqueté par ses dates de découverte et de fin de traitement. Les arcs de liaison sont ombrés. (b) Le graphe ${}^T G$, transposé de G . La forêt de parcours en profondeur calculée à la ligne 3 de COMPOSANTES-FORTEMENT-CONNEXES est représentée, avec les arcs de liaison ombrés. Chaque composante fortement connexe correspond à une arborescence de parcours en profondeur. Les sommets $b, c, g,$ et h , en gris foncé, sont les racines des arborescences de parcours en profondeur obtenus par le parcours en profondeur de ${}^T G$. (c) Le graphe sans circuit des composantes G^{CFC} obtenu par contraction de tous les arcs de chaque composante fortement connexe de G , de façon que chaque composante ne contienne plus qu'un seul sommet.

Notre algorithme de recherche des composantes fortement connexes d'un graphe $G = (S, A)$ utilise le transposé de G , défini à l'exercice 22.1.3 par ${}^T G = (S, {}^T A)$, où ${}^T A = \{(u, v) : (v, u) \in A\}$. Autrement dit, ${}^T A$ est constitué des arcs de G dont le sens a été inversé. Étant donné une représentation par listes d'adjacences de G ,

la création de ${}^T G$ demande un temps en $O(S + A)$. Il est intéressant d'observer que G et ${}^T G$ ont exactement les mêmes composantes fortement connexes : u et v sont accessibles dans G l'un à partir de l'autre si et seulement si ils le sont aussi dans ${}^T G$. La figure 22.9(b) montre le transposé du graphe de la figure 22.9(a) ; les composantes fortement connexes sont représentées en gris.

L'algorithme à temps linéaire (c'est-à-dire en $\Theta(S + A)$) que voici calcule les composantes fortement connexes d'un graphe orienté $G = (S, A)$ à l'aide de deux parcours en profondeur, l'un sur G et l'autre sur ${}^T G$.

COMPOSANTES-FORTEMENT-CONNEXES(G)

- 1 appeler $PP(G)$ pour calculer les dates de fin de traitement $f[u]$ pour chaque sommet u
- 2 calculer ${}^T G$
- 3 appeler $PP({}^T G)$, mais dans la boucle principale de PP , traiter les sommets par ordre de $f[u]$ (calculés en ligne 1) décroissants
- 4 imprimer les sommets de chaque arborescence de la forêt obtenue en ligne 3 en tant que composante fortement connexe distincte

L'idée sous-jacente à cet algorithme vient d'une propriété majeure du **graphe des composantes** $G^{CFC} = (S^{CFC}, A^{CFC})$, que nous définissons comme suit. Supposons que G ait des composantes fortement connexes C_1, C_2, \dots, C_k . L'ensemble des sommets S^{CFC} est $\{v_1, v_2, \dots, v_k\}$, et il contient un sommet v_i pour chaque composante fortement connexe C_i de G . Il y a un arc $(v_i, v_j) \in A^{CFC}$ si G contient un arc (x, y) pour un certain $x \in C_i$ et un certain $y \in C_j$. Vu d'une autre façon, en contractant tous les arcs dont les sommets incidents sont à l'intérieur de la même composante fortement connexe de G , le graphe obtenu est G^{CFC} . La figure 22.9(c) montre le graphe des composantes du graphe de la figure 22.9(a).

La propriété majeure est que le graphe des composantes est un graphe orienté sans circuit, ce qu'implique le lemme suivant.

Lemme 22.13 *Soient C et C' des composantes fortement connexes distinctes du graphe orienté $G = (S, A)$, soit $u, v \in C$, soit $u', v' \in C'$, et supposons qu'il y ait un chemin $u \rightsquigarrow u'$ dans G . Alors, il ne peut pas y avoir aussi un chemin $v' \rightsquigarrow v$ dans G .*

Démonstration : S'il y a un chemin $v' \rightsquigarrow v$ dans G , alors il y a des chemins $u \rightsquigarrow u' \rightsquigarrow v'$ et $v' \rightsquigarrow v \rightsquigarrow u$ dans G . Donc, u et v' sont accessibles mutuellement, ce qui contredit l'hypothèse selon laquelle C et C' sont des composantes fortement connexes distinctes. \square

Nous allons voir que le fait de traiter, dans le second parcours en profondeur, les sommets dans l'ordre décroissant des dates de fin de traitement calculées lors du premier parcours en profondeur, revient fondamentalement à visiter les sommets

du graphe des composantes (dont chacun correspond à une composante fortement connexe de G) dans un ordre topologique.

Comme COMPOSANTES-FORTEMENT-CONNEXES effectue deux parcours en profondeur, il risque d'y avoir ambiguïté quand on étudie $d[u]$ ou $f[u]$. Dans cette section, ces valeurs désignent systématiquement les dates de découverte et de fin de traitement calculées par le premier appel à PP en ligne 1.

Nous allons étendre la notation des dates de découverte et de fin de traitement à des ensembles de sommets. Si $U \subseteq S$, on définit

$$d(U) = \min_{u \in U} \{d[u]\} \quad \text{et} \quad f(U) = \max_{u \in U} \{f[u]\}$$

En d'autres termes, $d(U)$ et $f(U)$ désignent respectivement la date de découverte la plus précoce et la date de fin de traitement la plus tardive d'un sommet de U .

Le lemme suivant et son corollaire donnent une propriété fondamentale qui relie composantes fortement connexes et dates de fin de traitement dans le premier parcours en profondeur.

Lemme 22.14 *Soient C et C' des composantes fortement connexes distinctes d'un graphe orienté $G = (S, A)$. On suppose qu'il y a un arc $(u, v) \in A$, tel que $u \in C$ et $v \in C'$. Alors, $f(C) > f(C')$.*

Démonstration : Il y a deux cas, selon que c'est la composante fortement connexe C ou C' qui a eu le premier sommet découvert lors du parcours en profondeur.

Si $d(C) < d(C')$, soit x le premier sommet découvert dans C . À l'instant $d[x]$, tous les sommets de C et C' sont blancs. Il y a un chemin dans G entre x et chaque sommet de C , composé uniquement de sommets blancs. Comme $(u, v) \in A$, pour chaque sommet $w \in C'$, il y a aussi un chemin à l'instant $d[x]$ entre x et w dans G qui est composé uniquement de sommets blancs : $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. D'après le théorème du chemin blanc, tous les sommets de C et C' deviennent des descendants de x dans le parcours en profondeur. D'après le corollaire 22.8, $f[x] = f(C) > f(C')$.

Si l'on a, à la place, $d(C) > d(C')$, soit y le premier sommet découvert dans C' . À l'instant $d[y]$, tous les sommets de C' sont blancs et il y a un chemin dans G reliant y à chaque sommet de C' qui est composé uniquement de sommets blancs. D'après le théorème du chemin blanc, tous les sommets de C' deviennent des descendants de y dans l'arborescence de parcours en profondeur, et d'après le corollaire 22.8, $f[y] = f(C')$. À l'instant $d[y]$, tous les sommets de C sont blancs. Comme il y a un arc (u, v) entre C et C' , le lemme 22.13 implique qu'il ne peut pas y avoir de chemin entre C' et C . Donc, aucun sommet de C n'est accessible depuis y . À l'instant $f[y]$, par conséquent, tous les sommets de C sont encore blancs. Donc, pour tout sommet $w \in C$, on a $f[w] > f[y]$, ce qui entraîne que $f(C) > f(C')$. \square

Le corollaire suivant dit que chaque arc de ${}^T G$ qui relie des composantes fortement connexes différentes part d'une composante ayant une date de fin de traitement plus précoce (dans la première recherche en profondeur) et arrive à une composante ayant une date de fin de traitement plus tardive.

Corollaire 22.15 Soient C et C' des composantes fortement connexes distinctes d'un graphe orienté $G = (S, A)$. Supposons qu'il y ait un arc $(u, v) \in {}^T A$, tel que $u \in C$ et $v \in C'$. Alors, $f(C) < f(C')$.

Démonstration : Comme $(u, v) \in {}^T A$, on a $(v, u) \in A$. Comme les composantes fortement connexes de G et ${}^T G$ sont les mêmes, le lemme 22.14 entraîne que $f(C) < f(C')$. \square

Le corollaire 22.15 permet de comprendre le fonctionnement de COMPOSANTES-FORTEMENT-CONNEXES. Examinons ce qui se passe quand on fait le second parcours en profondeur, qui concerne ${}^T G$. On commence par la composante fortement connexe C dont la date de fin de traitement $f(C)$ est maximale. Le parcours part d'un certain sommet $x \in C$, et il visite tous les sommets de C . D'après le corollaire 22.15, il n'y a pas d'arc dans ${}^T G$ qui relie C à une autre composante fortement connexe, et donc la recherche issue de x ne visitera pas les sommets d'une quelconque autre composante. Par conséquent, l'arborescence de racine x contient les sommets de C , ni plus ni moins. Ayant fini de visiter tous les sommets de C , la recherche en ligne 3 sélectionne comme racine un sommet d'une certaine autre composante fortement connexe C' dont la date de fin de traitement $f(C')$ est maximale sur l'ensemble de toutes les composantes autres que C . Ici aussi, la recherche visitera tous les sommets de C' , mais d'après le corollaire 22.15, les seuls arcs de ${}^T G$ qui relient C' à une autre composante doivent aller vers C , que l'on a déjà visité. Plus généralement, quand le parcours en profondeur de ${}^T G$ en ligne 3 visite une composante fortement connexe, tous les arcs partant de cette composante mènent forcément vers des composantes qui ont déjà été visitées. Chaque arborescence de parcours en profondeur, par conséquent, est une seule composante fortement connexe. Le théorème suivant formalise cette démonstration.

Théorème 22.16 COMPOSANTES-FORTEMENT-CONNEXES(G) calcule effectivement les composantes fortement connexes d'un graphe orienté G .

Démonstration : Nous allons montrer, par récurrence sur le nombre d'arborescences de parcours en profondeur trouvées lors du parcours en profondeur de ${}^T G$ en ligne 3, que les sommets de chaque arborescence forment une composante fortement connexe. L'hypothèse de récurrence est que les k premières arborescences produisent en ligne 3 sont des composantes fortement connexes. La base de la récurrence, quand $k = 0$, est triviale.

Dans l'étape inductive, on suppose que chacun des k premières arborescences de parcours en profondeur produisent en ligne 3 est une composante fortement connexe, et l'on considère la $(k + 1)$ ème arborescence produite. Supposons que la racine de cette arborescence soit le sommet u , et que u soit dans la composante fortement connexe C . Compte tenu de la façon dont on a choisi les racines dans le parcours en profondeur en ligne 3, $f[u] = f(C) > f(C')$ pour toute composante fortement connexe C' autre que C qui reste à visiter. D'après l'hypothèse de récurrence, à l'instant où la recherche visite u , tous les autres sommets de C sont blancs. D'après le théorème du chemin blanc, par conséquent, tous les autres sommets de C sont des descendants de u dans

son arborescence de parcours en profondeur. En outre, d'après l'hypothèse de récurrence et le corollaire 22.15, tous les arcs de ${}^T G$ qui partent de C mènent forcément vers des composantes fortement connexes ayant été déjà visitées. Donc, aucun sommet d'une composante fortement connexe autre que C ne sera un descendant de u lors du parcours en profondeur de ${}^T G$. Donc, les sommets de l'arborescence de parcours en profondeur de ${}^T G$ qui a pour racine u forment une et une seule composante fortement connexe, ce qui termine l'étape inductive et la démonstration. \square

Voici une autre façon de comprendre le fonctionnement du second parcours en profondeur. Soit le graphe des composantes $({}^T G)^{CFC}$ de ${}^T G$. Si l'on associe chaque composante fortement connexe visitée dans le second parcours en profondeur à un sommet de $({}^T G)^{CFC}$, les sommets de $({}^T G)^{CFC}$ sont visités dans l'ordre topologique inverse. Si l'on inverse les arcs de $({}^T G)^{CFC}$, on obtient le graphe $T(({}^T G)^{CFC})$. Comme $T(({}^T G)^{CFC}) = G^{CFC}$ (voir exercice 22.5.4), le second parcours en profondeur visite les sommets de G^{CFC} dans l'ordre topologique.

Exercices

22.5.1 En quoi le nombre de composantes fortement connexes d'un graphe peut-il être modifié par l'ajout d'un nouvel arc ?

22.5.2 Décrire le fonctionnement de la procédure COMPOSANTES-FORTEMENT-CONNEXES sur le graphe de la figure 22.6. Plus précisément, donner les dates de fin de traitement calculées à la ligne 1 et la forêt obtenue à la ligne 3. On suppose que la boucle des lignes 5–7 de PP considère les sommets par ordre alphabétique et que les listes d'adjacences sont triées par ordre alphabétique.

22.5.3 Le professeur Kung-Fou affirme que l'algorithme des composantes fortement connexes peut être simplifié si l'on utilise le graphe initial (au lieu du transposé) lors du second parcours en profondeur et si l'on traite les sommets par ordre *croissant* des dates de fin de traitement. Le professeur a-t-il raison ?

22.5.4 Prouver que, pour tout graphe orienté G , on a $T(({}^T G)^{CFC}) = G^{CFC}$. Autrement dit, le transposé du graphe des composantes de ${}^T G$ est égal au graphe des composantes de G .

22.5.5 Donner un algorithme à temps $O(S+A)$ pour calculer le graphe des composantes d'un graphe orienté $G = (S, A)$. Vérifier qu'il existe au plus un arc entre deux sommets du graphe des composantes produit par votre algorithme.

22.5.6 Étant donné un graphe orienté $G = (S, A)$, expliquer comment créer un autre graphe $G' = (S, A')$ tel que (a) G' ait les mêmes composantes fortement connexes que G , (b) G' ait le même graphe des composantes que G et (c) A' soit le plus petit possible. Décrire un algorithme rapide permettant de calculer G' .

22.5.7 Un graphe orienté $G = (S, A)$ est dit *semi-connexe* si, pour toutes les paires de sommets $u, v \in S$, on a $u \rightsquigarrow v$ ou $v \rightsquigarrow u$. Donner un algorithme efficace qui détermine si G est ou non semi-connexe. Prouver sa validité et analyser son temps d'exécution.

PROBLÈMES

22.1. Classification des arcs par un parcours en largeur

Une forêt de parcours en profondeur classe les arcs d'un graphe en quatre catégories : arcs de liaison, arcs arrières, arcs avant et arcs transverses. Un arbre de parcours en largeur peut aussi servir à classer les arcs accessibles depuis l'origine du parcours dans les mêmes catégories.

- a.** Démontrer que, dans un parcours en largeur d'un graphe non orienté, les propriétés suivantes sont vérifiées :
- 1) Il n'existe aucun arc arrière, ni aucun arc avant.
 - 2) Pour chaque arc de liaison (u, v) , on a $d[v] = d[u] + 1$.
 - 3) Pour chaque arc transverse (u, v) , on a $d[v] = d[u]$ ou $d[v] = d[u] + 1$.
- b.** Démontrer que, lors du parcours en largeur d'un graphe orienté, les propriétés suivantes sont vérifiées :
- 1) Il n'existe aucun arc avant.
 - 2) Pour chaque arc de liaison (u, v) , on a $d[v] = d[u] + 1$.
 - 3) Pour chaque arc transverse (u, v) , on a $d[v] \leq d[u] + 1$.
 - 4) Pour chaque arc arrière (u, v) , on a $0 \leq d[v] < d[u]$.

22.2. Sommets d'articulation, ponts, et composantes biconnexes

Soit $G = (S, A)$ un graphe non orienté connexe. Un *sommet d'articulation* de G est un sommet dont la suppression rend G non connexe. Un *pont* de G est une arête dont la suppression rend G non connexe. Une *composante biconnexe* de G est un ensemble maximal d'arêtes tel que deux arêtes quelconques de l'ensemble se trouvent sur un même cycle élémentaire. La figure 22.10 illustre ces définitions. On peut déterminer les sommets d'articulation, les ponts et les composantes biconnexes à l'aide d'un parcours en profondeur. Soit $G_\pi = (S, A_\pi)$ une arborescence de parcours en profondeur de G .

- a.** Démontrer que la racine de G_π est un sommet d'articulation de G si et seulement si elle possède au moins deux enfants dans G_π .
- b.** Soit v un sommet non racine de G_π . Démontrer que v est un sommet d'articulation de G si et seulement si v possède un enfant s tel qu'il n'existe aucun arc arrière partant de s ou d'un descendant de s et arrivant à un ancêtre propre de v .

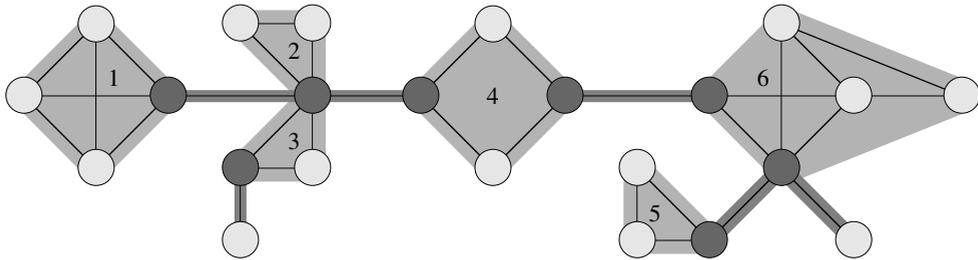


Figure 22.10 Les sommets d’articulation, les ponts et les composantes biconnexes d’un graphe non orienté connexe, servant de support au problème 22.5.9. Les sommets d’articulation sont les sommets foncés, les ponts sont les arêtes foncées et les composantes biconnexes sont les arêtes des régions en gris qui sont étiquetées par une numérotation cbc .

c. Soit

$$bas[v] = \min \left\{ \begin{array}{l} d[v], \\ \{d[w] : (u, w) \text{ est un arc arrière} \\ \text{pour un certain descendant } u \text{ de } v\} \end{array} \right.$$

Montrer comment calculer $bas[v]$ pour tous les sommets $v \in S$ en temps $O(A)$.

- d. Montrer comment calculer tous les sommets d’articulation en temps $O(A)$.
- e. Démontrer qu’une arête de G est un pont si et seulement si il n’appartient à aucun cycle élémentaire de G .
- f. Montrer comment calculer tous les ponts de G en temps $O(A)$.
- g. Démontrer que les composantes biconnexes de G partitionnent les arêtes de G qui ne sont pas des ponts.
- h. Donner un algorithme à temps $O(A)$ pour étiqueter chaque arête a de G par un entier positif $cbc[a]$ tel que $cbc[a] = cbc[a']$ si et seulement si a et a' se trouvent dans la même composante biconnexe.

22.3. Circuit eulérien

Dans un graphe orienté connexe $G = (S, A)$, un **circuit eulérien** est un circuit qui traverse chaque arc de G une fois exactement, bien qu’il puisse visiter un même sommet plus d’une fois.

a. Montrer que G contient un circuit eulérien si et seulement si

$$\text{degré-entrant}(v) = \text{degré-sortant}(v)$$

pour chaque sommet $v \in S$.

b. Décrire un algorithme en temps $O(A)$ pour trouver un circuit eulérien de G s’il en existe un. (*Conseil* : Fusionner les circuits à arcs disjoints.)

22.4. Accessibilité

Soit $G = (S, A)$ un graphe orienté dans lequel chaque sommet $u \in S$ est étiqueté par un entier unique $L(u)$ pris dans l'ensemble $\{1, 2, \dots, |S|\}$. Pour chaque sommet $u \in S$, soit $R(u) = \{v \in S : u \rightsquigarrow v\}$ l'ensemble des sommets qui sont accessibles depuis u . Définir $\min(u)$ comme étant le sommet de $R(u)$ dont l'étiquette est minimale ; c'est-à-dire que $\min(u)$ est le sommet v tel que $L(v) = \min \{L(w) : w \in R(u)\}$. Donner un algorithme à temps $O(S + A)$ qui calcule $\min(u)$ pour tous les sommets $u \in S$.

NOTES

Even [87] et Tarjan [292] constituent d'excellentes références sur les algorithmes de graphe.

Le parcours en largeur a été découvert par Moore [226] pour chercher des chemins dans des labyrinthes. Lee [198] découvrit indépendamment le même algorithme pour des problèmes de câblage sur les circuits imprimés.

Hopcroft et Tarjan [154] ont préconisé l'usage de la représentation par listes d'adjacences de préférence à la représentation par matrice d'adjacences pour les graphes peu denses, et ils furent les premiers à reconnaître l'importance algorithmique du parcours en profondeur. Le parcours en profondeur a été largement utilisé depuis la fin des années 1950, notamment dans les programmes d'intelligence artificielle.

Tarjan [289] proposa un algorithme à temps linéaire pour la recherche des composantes fortement connexes. L'algorithme de la section 22.5 traitant des composantes fortement connexes est adapté de Aho, Hopcroft et Ullman [6], qui en attribuent la paternité à S. R. Kosaraju (non publié) et M. Sharir [276]. Gabow [101] a, lui aussi, inventé un algorithme pour composantes fortement connexes, lequel utilise la contraction des circuits et emploie deux piles pour pouvoir s'exécuter en temps linéaire.

Knuth [182] fut le premier à donner un algorithme à temps linéaire pour le tri topologique.

Chapitre 23

Arbres couvrants de poids minimum

Lors de la phase de conception de circuits électroniques, on a souvent besoin de relier entre elles les broches de composants électriquement équivalents. Pour interconnecter un ensemble de n broches, on peut utiliser un arrangement de $n - 1$ branchements, chacun reliant deux broches. Parmi tous les arrangements possibles, celui qui utilise une longueur de branchements minimale est souvent le plus souhaitable.

On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe $G = (S, A)$ où S représente l'ensemble des broches, où A l'ensemble des interconnexions possibles entre paires de broches, et où pour chaque arête $(u, v) \in A$, on a un poids $w(u, v)$ qui spécifie le coût (longueur de fil nécessaire) pour connecter u et v . On souhaite alors trouver un sous-ensemble acyclique $T \subseteq A$ qui connecte tous les sommets et dont le poids total

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

soit minimum. Puisque T est acyclique et connecte tous les sommets, il doit former un arbre, que l'on appelle *arbre couvrant* car il « couvre » le graphe G . Le problème de la détermination de l'arbre T porte le nom de *problème de l'arbre couvrant minimal*.⁽¹⁾ La figure 23.1 montre un exemple de graphe connexe, avec son arbre couvrant minimal.

(1) L'expression « arbre couvrant minimal » est un raccourci pour « arbre couvrant de poids minimum ». Nous n'essayons pas, par exemple, de minimiser le nombre d'arêtes de T , puisque tous les arbres couvrants comportent exactement $|S| - 1$ arêtes d'après le théorème B.2.

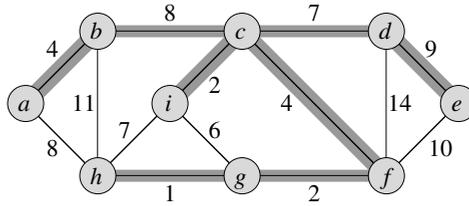


Figure 23.1 Un arbre couvrant minimum pour un graphe connexe. Les poids des arêtes sont représentés, et les arêtes de l'arbre couvrant minimum sont sur fond gris. Le poids total de l'arbre montré est 37. L'arbre couvrant minimal n'est pas unique : si l'on remplace l'arête (b, c) par l'arête (a, h) , on obtient un autre arbre couvrant de poids 37.

Dans ce chapitre, nous examinerons deux algorithmes permettant de résoudre le problème de l'arbre couvrant minimum : l'algorithme de Kruskal et l'algorithme de Prim. On peut aisément faire en sorte qu'ils s'exécutent chacun en $O(A \lg S)$, en utilisant des tas binaires ordinaires. En utilisant des tas de Fibonacci, l'algorithme de Prim peut être accéléré pour atteindre un temps d'exécution en $O(A + S \lg S)$, ce qui constitue une amélioration quand $|S|$ est très inférieur à $|A|$.

Les deux algorithmes sont des algorithmes gloutons (voir chapitre 16). À chaque étape de l'algorithme, une option parmi plusieurs possibles doit être choisie. La stratégie gloutonne effectue le choix qui semble le meilleur à l'instant donné. Une telle stratégie n'aboutit pas forcément à des solutions globalement optimales. Toutefois, pour le problème de l'arbre couvrant minimum, on peut démontrer que certaines stratégies gloutonnes génèrent effectivement un arbre couvrant de poids minimum. Bien que le présent chapitre puisse être lu indépendamment du chapitre 16, les méthodes gloutonnes présentées ici constituent une application classique des notions théoriques traitées dans ce chapitre antérieur.

La section 23.1 introduit un algorithme « générique » d'arbre couvrant minimum, qui fait croître un arbre couvrant en lui ajoutant une arête à la fois. La section 23.2 donne deux façons d'implémenter l'algorithme générique. Le premier algorithme, dû à Kruskal, est similaire à l'algorithme de composantes connexes de la section 21.1. Le second, dû à Prim, ressemble à l'algorithme de Dijkstra pour les plus courts chemins (section 24.3).

23.1 CONSTRUCTION D'UN ARBRE COUVRANT MINIMUM

Supposons qu'on dispose d'un graphe $G = (S, A)$ non orienté connexe avec une fonction de pondération $w : A \rightarrow \mathbf{R}$, et qu'on souhaite trouver un arbre couvrant minimum pour G . Les deux algorithmes étudiés dans ce chapitre utilisent une approche gloutonne, bien qu'elle soit appliquée différemment par chacun d'eux.

Cette stratégie gloutonne est explicitée par l'algorithme « générique » suivant, qui fait pousser l'arbre couvrant minimum arête par arête. L'algorithme

gère un ensemble d'arêtes E , en conservant l'invariant de boucle que voici :
avant chaque itération, E est un sous-ensemble d'un arbre couvrant minimum.

À chaque étape, on détermine une arête (u, v) qui peut être ajoutée à E tout en respectant cet invariant, au sens où $E \cup \{(u, v)\}$ est également un sous-ensemble d'un arbre couvrant minimum. On appelle ce type d'arête *arête sûre* pour E , car on peut l'ajouter à E sans détruire l'invariant.

ACM-GÉNÉRIQUE(G, w)

- 1 $E \leftarrow \emptyset$
- 2 **tant que** E ne forme pas un arbre couvrant
- 3 **faire** trouver une arête (u, v) qui est sûre pour E
- 4 $E \leftarrow E \cup \{(u, v)\}$
- 5 **retourner** E

Nous utilisons l'invariant de la façon que voici :

Initialisation : Après la ligne 1, l'ensemble E satisfait de manière triviale à l'invariant.

Conservation : La boucle des lignes 2–4 conserve l'invariant en ajoutant uniquement des arêtes sûres.

Terminaison : Toutes les arêtes ajoutées à E étant dans un arbre couvrant minimum, l'ensemble E retourné en ligne 5 est forcément un arbre couvrant minimum.

La partie délicate consiste, bien évidemment, à trouver une arête sûre en ligne 3. Il doit en exister une puisque, quand on exécute la ligne 3, l'invariant impose qu'il y ait un arbre couvrant T tel que $E \subseteq T$. Dans le corps de la boucle **tant que**, E doit être un sous-ensemble distinct de T ; par conséquent, il doit y avoir une arête $(u, v) \in T$ tel que $(u, v) \notin E$ et tel que (u, v) est sûre pour E .

Plus loin, nous donnons une règle (théorème 23.1) qui sert à reconnaître les arêtes sûres. La prochaine section décrit deux algorithmes qui ont recours à cette règle pour trouver efficacement des arêtes sûres.

Nous avons besoin de quelques définitions. La *coupe* $(P, S - P)$ d'un graphe non orienté $G = (S, A)$ est une partition de S . La figure 23.2 illustre cette notion. On dit qu'une arête $(u, v) \in A$ *traverse* la coupe $(P, S - P)$ si l'une de ses extrémités est un sommet de P et l'autre un sommet de $S - P$. On dit qu'une coupe *respecte* un ensemble E d'arêtes si aucune arête de E ne traverse la coupe. Une arête est une *arête minimale* pour la traversée de la coupe si son poids est minimal parmi toutes les arêtes qui traversent la coupe. Notez qu'il peut exister plus d'une arête minimale traversant une coupe si plusieurs poids sont égaux. Plus généralement, on dit qu'une arête est une *arête minimale* pour une propriété donnée, si son poids est minimal parmi tous les arêtes qui vérifient cette propriété.

Notre règle de reconnaissance des arêtes sûres est donnée par le théorème suivant.

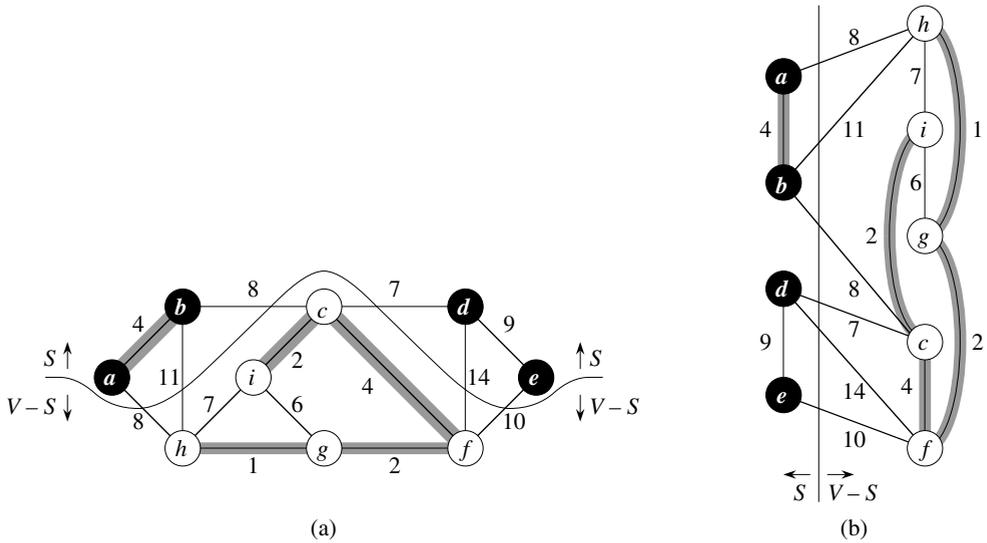


Figure 23.2 Deux manières de voir une coupe $(P, S - P)$ du graphe de la figure 23.1. **(a)** Les sommets de l'ensemble P sont représentés en noir, et ceux de $S - P$ en blanc. Les arêtes qui traversent la coupe sont ceux qui relient un sommet blanc à un sommet noir. L'arête (d, c) est la seule arête minimale qui traverse la coupe. Un sous-ensemble E d'arêtes est ombré ; notez que la coupe $(P, S - P)$ respecte E , car aucune arête de E ne traverse la coupe. **(b)** Le même graphe avec les sommets de l'ensemble P à gauche, et ceux de l'ensemble $S - P$ à droite. Une arête traverse la coupe si elle relie un sommet de gauche à un sommet de droite.

Théorème 23.1 Soit $G = (S, A)$ un graphe non orienté connexe, avec une fonction de pondération w à valeurs réelles définie sur A . Soit E un sous-ensemble de A inclus dans un arbre couvrant minimum de G , soit $(P, S - P)$ une coupe de G qui respecte E , et soit (u, v) une arête minimale traversant $(P, S - P)$. Alors, (u, v) est une arête sûre pour E .

Démonstration : Soit T un arbre couvrant minimum qui inclut E ; supposons que T ne contienne pas l'arête minimale (u, v) (si tel était le cas, la démonstration serait terminée). Nous allons construire un autre arbre couvrant minimum T' qui inclut $E \cup \{(u, v)\}$ en employant une technique de couper-coller, montrant ainsi que (u, v) est un arête sûre pour E .

L'arête (u, v) forme un cycle dans T avec les arêtes du chemin p de u à v , comme le montre la figure 23.3. Puisque u et v se trouvent de chaque côté de la coupe $(P, S - P)$, il existe au moins une arête de T sur le chemin p qui traverse aussi la coupe. Soit (x, y) une telle arête. L'arête (x, y) n'est pas dans E , car la coupe respecte E . Comme (x, y) se trouve sur le chemin unique de u à v dans T , la suppression de (x, y) sépare T en deux composantes. L'ajout de (u, v) les réassemble pour former un nouvel arbre couvrant

$$T' = T - \{(x, y)\} \cup \{(u, v)\}.$$

Montrons à présent que T' est un arbre couvrant minimum. Puisque (u, v) est une arête minimale traversant $(P, S - P)$ et que (x, y) traverse également cette coupe, $w(u, v) \leq w(x, y)$.

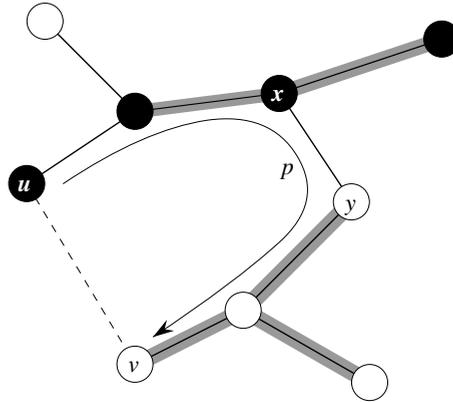


Figure 23.3 La preuve du théorème 23.1. Les sommets de P sont noirs, et les sommets de $S - P$ sont blancs. Les arêtes de l'arbre couvrant minimum T sont représentées, mais ceux du graphe G ne le sont pas. Les arêtes de E sont ombrées, et (u, v) est une arête minimale traversant la coupe $(P, S - P)$. L'arête (x, y) est une arête du chemin p unique de u à v dans T . Un arbre couvrant minimum T' qui contient (u, v) est formé en supprimant l'arête (x, y) de T et en ajoutant l'arête (u, v) .

Donc,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) . \end{aligned}$$

Mais T est un arbre couvrant minimum, de sorte que $w(T) \leq w(T')$; T' est donc lui aussi un arbre couvrant minimum.

Il reste à montrer que (u, v) est effectivement une arête sûre pour E . On a $E \subseteq T'$, car $E \subseteq T$ et $(x, y) \notin E$; donc, $E \cup \{(u, v)\} \subseteq T'$. Par suite, comme T' est un arbre couvrant minimum, (u, v) est une arête sûre pour E . \square

Le théorème 23.1 permet de mieux comprendre le comportement de l'algorithme ACM-GÉNÉRIQUE sur un graphe connexe $G = (S, A)$. Pendant l'exécution, l'ensemble E est toujours acyclique ; sinon, un arbre couvrant minimum incluant E contiendrait un cycle, ce qui amène une contradiction. À un moment quelconque de l'exécution, le graphe $G_E = (S, E)$ est une forêt et chacune des composantes connexes de G_E est un arbre. (Certains arbres peuvent ne contenir qu'un seul sommet, comme c'est le cas par exemple au commencement de l'algorithme : E est vide et la forêt contient $|S|$ arbres, un pour chaque sommet.) Par ailleurs, n'importe quelle arête sûre (u, v) pour E relie des composantes distinctes de G_E , puisque $E \cup \{(u, v)\}$ doit être acyclique.

La boucle des lignes 2–4 de ACM-GÉNÉRIQUE est exécutée $|S| - 1$ fois, car chacun des $|S| - 1$ arêtes d'un arbre couvrant minimum est successivement déterminé. Au départ, quand $E = \emptyset$, il existe $|S|$ arbres dans G_E , et chaque itération réduit ce nombre de 1. Quand la forêt ne contient plus qu'un seul arbre, l'algorithme se termine.

Le corollaire suivant du théorème 23.1 est utilisé par les deux algorithmes de la section 23.2.

Corollaire 23.2 Soit $G = (S, A)$ un graphe non orienté connexe, avec une fonction de pondération w à valeurs réelles définie sur A . Soit E un sous-ensemble de A inclus dans un arbre couvrant minimum de G , et soit C une composante connexe (arbre) de la forêt $G_E = (S, E)$. Si (u, v) est une arête minimale reliant C à une autre composante de G_E , alors (u, v) est une arête sûre pour E .

Démonstration : La coupe $(V_C, V - V_C)$ respecte E et (u, v) est une arête minimale pour cette coupe. Donc, (u, v) est sûre pour E . \square

Exercices

23.1.1 Soit (u, v) une arête de poids minimal dans un graphe G . Montrer que (u, v) appartient à un arbre couvrant minimum de G .

23.1.2 Le professeur Sabatier conjecture la réciproque du théorème 23.1. Soit $G = (S, A)$ un graphe non orienté connexe, avec une fonction de pondération à valeurs réelles w définie sur A . Soit E un sous ensemble de A qui est inclus dans un arbre couvrant minimum de G , soit $(P, S - P)$ une coupe de G qui respecte E , soit enfin (u, v) une arête sûre pour E traversant $(P, S - P)$. Alors, (u, v) est une arête minimale pour la coupe. Montrer que la conjecture du professeur est incorrecte, en donnant un contre-exemple.

23.1.3 Montrer que, si une arête (u, v) appartient à un arbre couvrant minimum, alors c'est une arête minimale traversant une coupe du graphe.

23.1.4 Donner un exemple simple de graphe connexe pour lequel l'ensemble des arêtes $\{(u, v) : \text{il existe une coupe } (S, V - S) \text{ telle que } (u, v) \text{ soit une arête minimale traversant } (S, V - S)\}$ ne forme pas un arbre couvrant minimum.

23.1.5 Soit a une arête de poids maximal d'un cycle de $G = (S, A)$. Démontrer qu'il existe un arbre couvrant minimum de $G' = (S, A - \{a\})$ qui est également un arbre couvrant minimum de G . En d'autres termes, il existe un arbre couvrant minimum de G qui ne contient pas a .

23.1.6 Montrer qu'un graphe possède un arbre couvrant minimum unique si, pour chaque coupe du graphe, il existe une arête minimale unique traversant cette coupe. Montrer que la réciproque n'est pas vraie, en donnant un contre-exemple.

23.1.7 Montrer que, si tous les poids d'arête d'un graphe sont positifs, alors un sous-ensemble d'arêtes qui relie tous les sommets et qui a un poids total minimal est forcément un arbre. Donner un exemple montrant que cette conclusion n'est plus valable si l'on autorise des poids négatifs.

23.1.8 Soit T un arbre couvrant minimum d'un graphe G , et soit L la liste triée des poids d'arêtes de T . Montrer que, pour tout autre arbre couvrant minimum T' de G , la liste L est également la liste triée des poids d'arête de T' .

23.1.9 Soit T un arbre couvrant minimum d'un graphe $G = (S, A)$, et soit S' un sous-ensemble de S . Soit T' le sous-graphe de T induit par S' , et soit G' le sous-graphe de G induit par S' . Montre que, si T' est connexe, alors T' est un arbre couvrant minimum de G' .

23.1.10 Étant donné un graphe G et un arbre couvrant minimum T , on suppose que l'on diminue le poids de l'une des arêtes de T . Montrer que T est encore un arbre couvrant minimum de G . De manière plus formelle, soit T un arbre couvrant minimum de G dont les poids d'arête sont donnés par la fonction w . Choisir une arête $(x, y) \in T$ et un nombre positif k , puis définir la fonction de pondération w' par

$$w'(u, v) = \begin{cases} w(u, v) & \text{si } (u, v) \neq (x, y), \\ w(x, y) - k & \text{si } (u, v) = (x, y). \end{cases}$$

Montrer que T est un arbre couvrant minimum de G si les poids d'arête sont donnés par w' .

23.1.11 * Étant donné un graphe G et un arbre couvrant minimum T , on suppose que l'on diminue le poids de l'une des arêtes qui n'est pas dans T . Donner un algorithme permettant de trouver l'arbre couvrant minimum du graphe ainsi modifié.

23.2 ALGORITHMES DE KRUSKAL ET DE PRIM

Les deux algorithmes d'arbre couvrant minimum décrits dans cette section sont élaborés à partir de l'algorithme générique. Ils utilisent chacun une règle spécifique pour déterminer l'arête sûre recherchée en ligne 3 de ACM-GÉNÉRIQUE. Dans l'algorithme de Kruskal, l'ensemble E est une forêt. L'arête sûre ajoutée à E est toujours une arête de moindre poids du graphe qui relie deux composantes distinctes. Dans l'algorithme de Prim, l'ensemble E est un arbre. L'arête sûre ajoutée à E est toujours une arête de moindre poids qui relie l'arbre à un sommet extérieur.

a) Algorithme de Kruskal

L'algorithme de Kruskal s'inspire directement de l'algorithme générique de l'arbre couvrant minimum donné à la section 23.1. Il trouve une arête sûre à ajouter à la forêt en cherchant, parmi toutes les arêtes reliant deux arbres quelconques de la forêt, une arête (u, v) de poids minimal. Soient C_1 et C_2 les deux arbres reliés par (u, v) . Puisque (u, v) doit être une arête minimale reliant C_1 à un autre arbre, le corollaire 23.2 implique que (u, v) est une arête sûre pour C_1 . L'algorithme de Kruskal est un algorithme glouton car, à chaque étape, il ajoute à la forêt une arête de poids minimal.

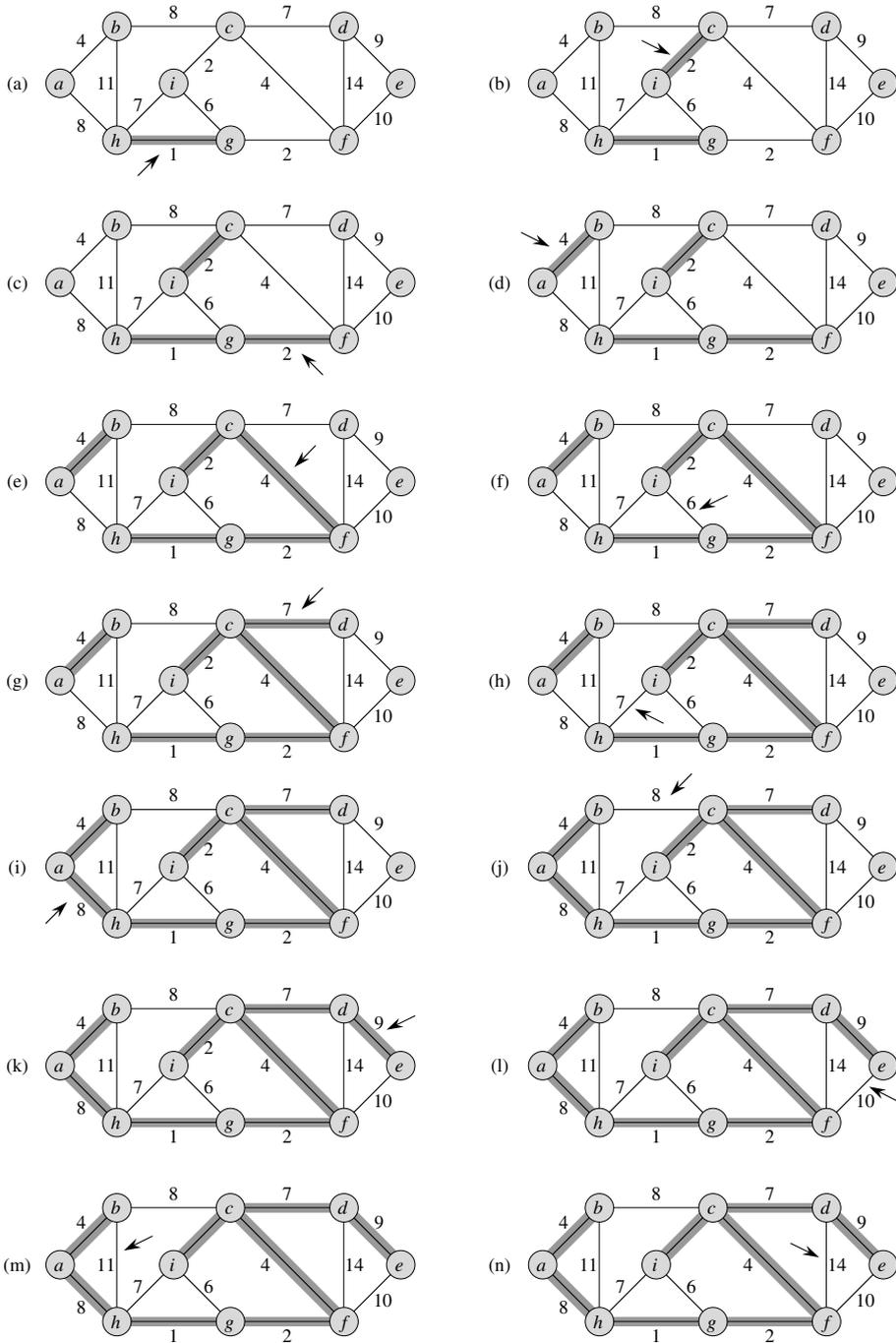


Figure 23.4 L'exécution de l'algorithme de Kruskal sur le graphe de la figure 23.1. Les arêtes sur fond gris appartiennent à la forêt E en cours de construction. Les arêtes sont considérées par l'algorithme par ordre de poids croissant. Une flèche indique l'arête sélectionnée à chaque étape de l'algorithme. Si l'arête relie deux arbres distincts de la forêt, elle est ajoutée à la forêt, provoquant alors la fusion des deux arbres.

Notre implémentation de l'algorithme de Kruskal ressemble à celle de l'algorithme de calcul des composantes connexes vu à la section 21.1. Elle utilise une structure de données d'ensembles disjoints pour gérer plusieurs ensembles disjoints. Chaque ensemble contient les sommets d'un arbre de la forêt courante. L'opération TROUVER-ENSEMBLE(u) retourne un élément représentatif de l'ensemble qui contient u . Ainsi, on peut déterminer si deux sommets u et v appartiennent au même arbre, en testant si TROUVER-ENSEMBLE(u) est égal à TROUVER-ENSEMBLE(v). La combinaison des arbres est effectuée par la procédure UNION.

ACM-KRUSKAL(G, w)

```

1   $E \leftarrow \emptyset$ 
2  pour chaque sommet  $v \in S[G]$ 
3      faire CRÉER-ENSEMBLE( $v$ )
4  trier les arêtes de  $A$  par ordre croissant de poids  $w$ 
5  pour chaque arête  $(u, v) \in A$  pris par ordre de poids croissant
6      faire si TROUVER-ENSEMBLE( $u$ )  $\neq$  TROUVER-ENSEMBLE( $v$ )
7          alors  $E \leftarrow E \cup \{(u, v)\}$ 
8              UNION( $u, v$ )
9  retourner  $E$ 
```

Le fonctionnement de l'algorithme de Kruskal est illustré à la figure 23.4. Les lignes 1–3 initialisent l'ensemble E à l'ensemble vide et créent $|S|$ arbres, un pour chaque sommet. Les arêtes de A sont triées par ordre de poids croissant en ligne 4. La boucle **pour** des lignes 5–8 teste, pour chaque arête (u, v) , si ses extrémités u et v appartiennent au même arbre. Si c'est le cas, l'arête (u, v) ne peut pas être ajoutée à la forêt sans créer de cycle ; l'arête est donc rejetée. Sinon, l'arête (u, v) est ajoutée à E en ligne 7, et les sommets des deux arbres sont fusionnés en ligne 8.

Le temps d'exécution de l'algorithme de Kruskal sur un graphe $G = (S, A)$ dépend de l'implémentation de la structure d'ensembles disjoints. Pour la suite, nous retiendrons l'implémentation par forêt d'ensembles disjoints, vue à la section 21.3, associée aux heuristiques de l'union par rang et de la compression de chemin, car elle est asymptotiquement la plus rapide connue à ce jour. L'initialisation de l'ensemble E en ligne 1 requiert un temps $O(1)$, et le temps pris par le tri des arcs en ligne 4 est $O(A \lg A)$. (Nous tiendrons compte du coût des $|S|$ opérations CRÉER-ENSEMBLE de la boucle **pour** des lignes 1–3 dans un moment.) La boucle **pour** des lignes 5–8 fait $O(A)$ opérations TROUVER-ENSEMBLE et UNION sur la forêt d'ensembles disjoints. Avec les $|S|$ opérations CRÉER-ENSEMBLE, cela fait un total de $O((S+A)\alpha(S))$ temps, où α est la fonction à très lente croissance définie à la section 21.4. Comme G est censé être connexe, on a $|A| \geq |S| - 1$, et donc les opérations d'ensembles disjoints prennent un temps $O(A \alpha(S))$. En outre, comme $\alpha(|S|) = O(\lg S) = O(\lg A)$, le temps d'exécution total de l'algorithme de Kruskal est $O(A \lg A)$. On observant que $|A| < |S|^2$, on a $\lg |A| = O(\lg S)$, et l'on peut donc reformuler le temps d'exécution de l'algorithme de Kruskal comme étant $O(A \lg S)$.

b) Algorithme de Prim

Comme l'algorithme de Kruskal, l'algorithme de Prim est un cas particulier de l'algorithme générique étudié à la section 23.1. L'algorithme de Prim ressemble à l'algorithme de Dijkstra de recherche des plus courts chemins d'un graphe, que nous verrons à la section 24.3. L'algorithme de Prim a pour propriété que les arêtes de l'ensemble E constituent toujours un arbre. Comme le montre la figure 23.5, l'arbre démarre d'un sommet racine r arbitraire puis croît jusqu'à couvrir tous les sommets de S . À chaque étape, une arête minimale est ajoutée à l'arbre E pour relier celui-ci à un sommet isolé de $G_E = (S, E)$. D'après le corollaire 23.2, cette règle permet de n'ajouter que des arêtes qui sont sûres pour E ; ainsi, quand l'algorithme se termine, les arêtes de E forment un arbre couvrant minimum. Cette stratégie est gloutonne puisque, à chaque étape, l'arbre est augmenté d'une arête qui accroît le moins possible le poids total de l'arbre.

Pour implémenter efficacement l'algorithme de Prim, l'important est de faciliter la sélection de la nouvelle arête à ajouter à l'arbre constitué des arêtes de E . Dans le pseudo code qui suit, le graphe connexe G et la racine r de l'arbre couvrant minimum à construire sont les entrées de l'algorithme. Pendant l'exécution, tous les sommets qui n'appartiennent pas à l'arbre se trouvent dans une file de priorités min F basée sur un champ *clé*. Pour chaque sommet v , $clé[v]$ est le poids minimal d'une arête reliant v à un sommet de l'arbre ; par convention, $clé[v] = \infty$ si une telle arête n'existe pas. Le champ $\pi[v]$ désigne le parent de v dans l'arbre.

Pendant le déroulement de l'algorithme, l'ensemble E de ACM-GÉNÉRIQUE est conservé implicitement sous la forme

$$E = \{(v, \pi[v]) : v \in S - \{r\} - F\} .$$

Quand l'algorithme se termine, la file de priorités min F est vide ; l'arbre couvrant minimum de G est donc

$$E = \{(v, \pi[v]) : v \in S - \{r\}\} .$$

ACM-PRIM(G, w, r)

```

1  pour chaque  $u \in S[G]$ 
2      faire  $clé[u] \leftarrow \infty$ 
3           $\pi[u] \leftarrow \text{NIL}$ 
4   $clé[r] \leftarrow 0$ 
5   $F \leftarrow S[G]$ 
6  tant que  $F \neq \emptyset$ 
7      faire  $u \leftarrow \text{EXTRAIRE-MIN}(F)$ 
8          pour chaque  $v \in \text{Adj}[u]$ 
9              faire si  $v \in F$  et  $w(u, v) < clé[v]$ 
10                 alors  $\pi[v] \leftarrow u$ 
11                  $clé[v] \leftarrow w(u, v)$ 

```

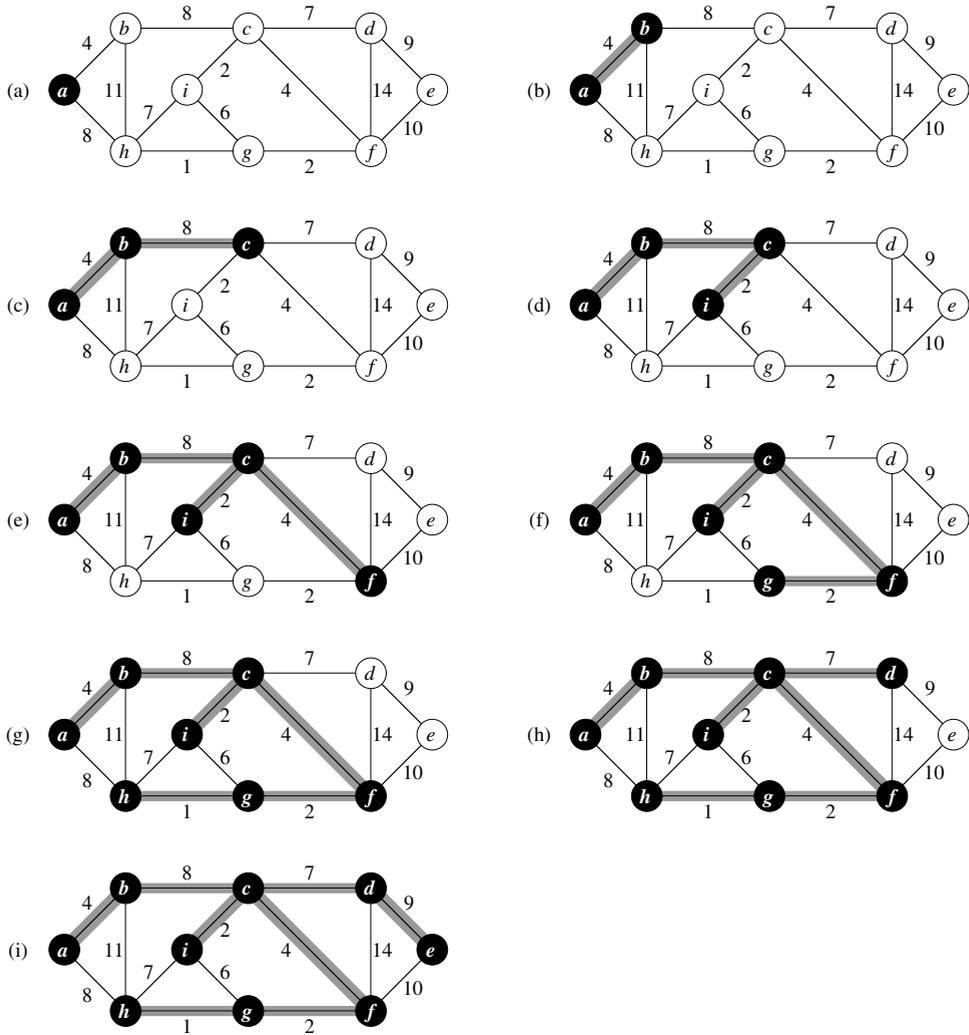


Figure 23.5 L'exécution de l'algorithme de Prim sur le graphe de la figure 23.1. Le sommet racine est *a*. Les arêtes sur fond gris appartiennent à l'arbre en cours de construction, et les sommets de l'arbre déterminent une coupe du graphe, et une arête minimale traversant la coupe est ajoutée à l'arbre. Dans la deuxième étape, par exemple, l'algorithme a le choix entre ajouter l'arête (*b, c*) ou l'arête (*a, h*) puisque toutes les deux sont des arêtes minimales traversant la coupe.

L'algorithme de Prim fonctionne comme illustré sur la figure 23.5. Les lignes 1–5 initialisent la clé de chaque sommet à ∞ (exception faite de la racine r , dont la clé est initialisée à 0 pour qu'elle soit le premier sommet traité), initialisent le parent de chaque sommet à NIL et initialisent la file de priorités F de façon qu'elle contienne tous les sommets. L'algorithme conserve l'invariant de boucle suivant, composé de trois parties : Avant chaque itération

de la boucle **tant que** des lignes 6–11,

- 1) $E = \{(v, \pi[v]) : v \in S - \{r\} - F\}$.
- 2) Les sommets déjà placés dans l'arbre couvrant minimum sont ceux de $S - F$.
- 3) Pour tous les sommets $v \in F$, si $\pi[v] \neq \text{NIL}$, alors $\text{clé}[v] < \infty$ et $\text{clé}[v]$ est le poids d'une arête minimale $(v, \pi[v])$ qui relie v à un certain sommet déjà placé dans l'arbre couvrant minimum.

La ligne 7 identifie un sommet $u \in Q$ qui est incident pour une arête minimale traversant la coupe $(S - F, F)$ (sauf dans la première itération, où $u = r$ à cause de la ligne 4). Supprimer u de l'ensemble F a pour effet de l'ajouter à l'ensemble $S - F$ des sommets de l'arbre, et donc d'ajouter $(u, \pi[u])$ à E . La boucle **pour** des lignes 8–11 met à jour les champs clé et π de chaque sommet v adjacent à u mais pas dans l'arbre. Cette mise à jour conserve la troisième partie de l'invariant.

Les performances de l'algorithme de Prim dépendent de la manière dont la file de priorités $\min F$ est implémentée. Si F est implémentée comme un tas \min binaire (voir chapitre 6), on peut se servir de la procédure CONSTRUIRE-TAS-MIN pour effectuer l'initialisation des lignes 1–5 en temps $O(S)$. Le corps de la boucle **tant que** est exécuté $|S|$ fois et comme chaque opération EXTRAIRE-MIN prend un temps $O(\lg S)$, le temps total requis par tous les appels à EXTRAIRE-MIN est $O(S \lg S)$. La boucle **pour** des lignes 8–11 est exécutée en tout $O(A)$ fois, puisque la somme des longueurs de toutes les listes d'adjacences vaut $2|A|$. À l'intérieur de la boucle **pour**, le test d'appartenance à F de la ligne 9 peut être implémenté en temps constant, via attribution à chaque sommet d'un bit indiquant s'il appartient ou non à F et mise à jour de ce bit quand le sommet est supprimé de F . L'affectation en ligne 11 implique une opération DIMINUER-CLÉ implicite sur le tas \min , qu'on peut implémenter dans un tas \min binaire en $O(\lg S)$. Donc, le temps total de l'algorithme de Prim est $O(S \lg S + A \lg S) = O(A \lg S)$, qui est asymptotiquement le même que pour notre implémentation de l'algorithme de Kruskal.

Cependant, le temps d'exécution asymptotique de l'algorithme de Prim peut être amélioré en utilisant des tas de Fibonacci. Le chapitre 20 montre que si $|S|$ éléments sont organisés en tas de Fibonacci, on peut effectuer une opération EXTRAIRE-MIN dans un temps amorti $O(\lg S)$ et une opération DIMINUER-CLÉ (pour implémenter la ligne 11) dans un temps amorti $O(1)$. En conséquence, si l'on se sert d'un tas de Fibonacci pour implémenter la file de priorités $\min F$, le temps d'exécution de l'algorithme de Prim est amélioré pour devenir $O(A + S \lg S)$.

Exercices

23.2.1 L'algorithme de Kruskal peut retourner des arbres couvrants différents pour un même graphe G , selon l'ordre dans lequel sont examinés les arêtes de même poids dans la liste triée. Montrer que, pour chaque arbre couvrant minimum T de G , il existe un moyen de trier les arêtes de G dans l'algorithme de Kruskal pour que l'algorithme retourne T .

23.2.2 On suppose que le graphe $G = (S, A)$ est représenté par une matrice d'adjacences. Donner une implémentation simple de l'algorithme de Prim pour ce cas, qui s'exécute en $O(S^2)$.

23.2.3 L'implémentation de l'algorithme de Prim via tas de Fibonacci est-elle asymptotiquement plus rapide que l'implémentation via tas binaire d'un graphe peu dense $G = (S, A)$, où $|A| = \Theta(S)$? Et pour un graphe dense, où $|A| = \Theta(S^2)$? Quelle doit être la relation entre $|A|$ et $|S|$ pour que l'implémentation via tas de Fibonacci soit asymptotiquement plus rapide que l'implémentation via tas binaire ?

23.2.4 On suppose que tous les poids des arêtes d'un graphe sont des entiers compris entre 1 et $|S|$. Quel est alors le temps d'exécution de l'algorithme de Kruskal ? Que se passe-t-il si les poids sont des entiers compris entre 1 et W , W étant une constante fixée ?

23.2.5 On suppose que tous les poids des arêtes d'un graphe sont des entiers compris entre 1 et $|S|$. Quel est alors le temps d'exécution de l'algorithme de Prim ? Que se passe-t-il si les poids sont des entiers compris entre 1 et W , W étant une constante fixée ?

23.2.6 * On suppose que les poids des arêtes d'un graphe sont uniformément répartis sur l'intervalle semi-ouvert $[0, 1[$. Lequel des deux algorithmes, celui de Kruskal ou celui de Prim, peut-on accélérer ?

23.2.7 * On suppose qu'un graphe G possède un arbre couvrant minimum déjà calculé. Quel est le temps requis pour la mise à jour de l'arbre couvrant minimum si un nouveau sommet et des arêtes incidentes sont ajoutées à G ?

23.2.8 Le professeur Bizard propose un nouvel algorithme diviser-pour-régner pour le calcul des arbres couvrants minimum, dont voici le principe. Étant donné un graphe $G = (S, A)$, on partitionne l'ensemble S des sommets en deux ensemble S_1 et S_2 tels que $|S_1|$ et $|S_2|$ diffèrent d'au plus 1. Soit A_1 l'ensemble des arêtes qui ne sont incidentes qu'à des sommets de S_1 et soit A_2 l'ensemble des arêtes qui ne sont incidentes qu'à des sommets de S_2 . On résout alors récursivement un problème d'arbre couvrant minimum pour chacun des deux sous-graphes $G_1 = (S_1, A_1)$ et $G_2 = (S_2, A_2)$. Enfin, on sélectionne l'arête de poids minimal de A qui traverse la coupe (S_1, S_2) et l'on utilise cette arête pour réunifier en un seul arbre couvrant les deux arbres couvrants minimum résultants. Prouver que cet algorithme calcule effectivement un arbre couvrant minimum pour G , ou bien donner un contre-exemple.

PROBLÈMES

23.1. Deuxième meilleur arbre couvrant minimum

Soit $G = (S, A)$ un graphe connexe non orienté avec une fonction de pondération $w : A \rightarrow \mathbf{R}$; on suppose que $|A| \geq |S|$ et que tous les poids des arêtes sont distincts.

Soit \mathcal{T} l'ensemble de tous les arbres couvrants de G et soit T' un arbre couvrant minimum de G . On appelle **deuxième meilleur arbre couvrant minimum** un arbre couvrant T tel que $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

- a. Montrer que l'arbre couvrant minimum est unique, mais que ce n'est pas forcément le cas du deuxième meilleur arbre couvrant minimum.
- b. Soit T un arbre couvrant minimum de G . Démontrer qu'il existe des arêtes $(u, v) \in T$ et $(x, y) \notin T$ tels que $T - \{(u, v)\} \cup \{(x, y)\}$ soit un deuxième meilleur arbre couvrant minimum de G .
- c. Soit T un arbre couvrant de G et, pour deux sommets $u, v \in S$ quelconques, soit $\max[u, v]$ une arête de poids maximal sur le chemin unique entre u et v dans T . Décrire un algorithme en $O(S^2)$ qui, à partir de T , calcule $\max[u, v]$ pour tout $u, v \in S$.
- d. Donner un algorithme efficace permettant de calculer le deuxième meilleur arbre couvrant minimum de G .

23.2. Arbre couvrant minimum pour des graphes peu denses

Pour un graphe connexe $G = (S, A)$ très peu dense, on peut améliorer le temps d'exécution $O(A + S \lg S)$ de l'algorithme de Prim avec des tas de Fibonacci en « pré traitant » G pour diminuer le nombre de sommets avant d'exécuter l'algorithme de Prim. En particulier on choisit, pour chaque sommet u , l'arête de poids minimal (u, v) incidente à u et on place (u, v) dans l'arbre couvrant minimum en cours de construction. On contracte alors toutes les arêtes choisies (voir section B.4). Au lieu de contracter ces arêtes une par une, on commence par identifier des ensembles de sommets qui seront combinés dans le même nouveau sommet. On crée alors le graphe qui aurait résulté de la contraction de ces arêtes une par une, mais on le fait en « renommant » les arêtes en fonction des ensembles dans lesquels leurs extrémités avaient été placées. Plusieurs arêtes du graphe originel risquent d'être renommées de la même façon. En pareil cas, on ne garde qu'une seule arête et son poids est le minimum des poids des arêtes originales correspondants.

Au début, l'arbre couvrant minimum T à construire est vide et, pour chaque arête $(u, v) \in E$, on fait $\text{orig}[u, v] = (u, v)$ et $c[u, v] = w(u, v)$. On utilise l'attribut orig pour référencer l'arc du graphe initial qui est associé à une arête du graphe contracté. L'attribut c contient le poids d'une arête et, à mesure que des arêtes sont contractées, il est mis à jour selon le mécanisme susmentionné de sélection des poids d'arête. La procédure ACM-RÉDUIRE a comme entrées G , orig , c et T ; elle retourne un graphe

contracté G' ainsi que des attributs actualisés $orig'$ et c' pour le graphe G' . La procédure, en outre, accumule des arêtes de G pour construire l'arbre couvrant minimum T .

```

ACM-RÉDUIRE( $G, orig, c, T$ )
1  pour chaque  $v \in S[G]$ 
2      faire  $marque[v] \leftarrow$  FAUX
3          CRÉER-ENSEMBLE( $v$ )
4  pour chaque  $u \in S[G]$ 
5      faire si  $marque[u] =$  FAUX
6          alors choisir  $v \in Adj[u]$  tel que  $c[u, v]$  soit minimisé
7              UNION( $u, v$ )
8               $T \leftarrow T \cup \{orig[u, v]\}$ 
9               $marque[u] \leftarrow marque[v] \leftarrow$  VRAI
10  $S[G'] \leftarrow \{\text{TROUVER-ENSEMBLE}(v) : v \in S[G]\}$ 
11  $A[G'] \leftarrow \emptyset$ 
12 pour chaque  $(x, y) \in A[G]$ 
13     faire  $u \leftarrow$  TROUVER-ENSEMBLE( $x$ )
14          $v \leftarrow$  TROUVER-ENSEMBLE( $y$ )
15         si  $(u, v) \notin A[G']$ 
16             alors  $A[G'] \leftarrow A[G'] \cup \{(u, v)\}$ 
17                  $orig'[u, v] \leftarrow orig[x, y]$ 
18                  $c'[u, v] \leftarrow c[x, y]$ 
19             sinon si  $c[x, y] < c'[u, v]$ 
20                 alors  $orig'[u, v] \leftarrow orig[x, y]$ 
21                      $c'[u, v] \leftarrow c[x, y]$ 
22 construire listes d'adjacences  $Adj$  pour  $G'$ 
23 retourner  $G', orig', c'$  et  $T$ 
    
```

- a. Soit T l'ensemble d'arcs retourné par ACM-RÉDUIRE ; soit E l'arbre couvrant minimum du graphe G' formé par l'appel ACM-PRIM(G', c', r), où r est un sommet quelconque de $S[G']$. Prouver que $T \cup \{orig'[x, y] : (x, y) \in E\}$ est un arbre couvrant minimum de G .
- b. Montrer que $|S[G']| \leq |S|/2$.
- c. Montrer comment implémenter ACM-RÉDUIRE pour qu'elle tourne en temps $O(A)$. (*Conseil* : Utiliser des structures de données simples.)
- d. Supposez que l'on exécute k phases de ACM-RÉDUIRE, les sorties $G', orig'$ et c' d'une phase devenant les entrées $G, orig$ et c de la phase suivante, et que l'on accumule les arêtes dans T . Montrer que le temps d'exécution global des k phases est $O(kA)$.
- e. Supposez que, après exécution de k phases de ACM-RÉDUIRE comme indiqué en partie (d), on exécute l'algorithme de Prim en appelant ACM-PRIM(G', c', r), où G' et c' sont retournés par la dernière phase et où r est un sommet quelconque de $S[G']$. Montrer comment choisir k pour que le temps d'exécution global soit

$O(A \lg \lg S)$. Prouver que votre choix de k minimise le temps d'exécution asymptotique global.

- f. Pour quelles valeurs de $|A|$ (exprimées en fonction de $|S|$) l'algorithme de Prim avec pré traitement l'emporte-t-il asymptotiquement sur l'algorithme de Prim ordinaire ?

23.3. Arbre couvrant à goulet d'étranglement

Un *arbre couvrant à goulet d'étranglement* T d'un graphe non orienté G est un arbre couvrant de G dont le poids maximal d'une arête est minimum par rapport à l'ensemble des tous les arbres couvrants de G . On dira que la valeur de l'arbre couvrant à goulet d'étranglement est le poids de l'arête de poids maximal de T .

- a. Montrer qu'un arbre couvrant minimum est un arbre couvrant à goulet d'étranglement.

La partie (a) montre que trouver un arbre couvrant à goulet d'étranglement n'est pas plus difficile que de trouver un arbre couvrant minimum. Dans les parties qui suivent, nous allons montrer qu'on peut en trouver un en temps linéaire.

- b. Donner un algorithme à temps linéaire qui, à partir d'un graphe G et d'un entier b , détermine si la valeur de l'arbre couvrant à goulet d'étranglement est au plus égale à b .
- c. Utiliser l'algorithme de la partie (b) comme sous-routine d'un algorithme à temps linéaire pour le problème de l'arbre couvrant à goulet d'étranglement. (*Conseil* : Vous aurez peut-être besoin d'une sous-routine qui contracte des ensembles d'arcs, comme dans la procédure ACM-RÉDUIRE du problème 23.2.)

23.4. Autres algorithmes d'arbre couvrant minimum

Dans ce problème, on va donner le pseudo code de trois autres algorithmes. Chacun d'eux prend un graphe en entrée et retourne un ensemble d'arêtes T . Pour chaque algorithme, il faut montrer soit que T est un arbre couvrant minimum, soit que ce n'en est pas un. Il faut aussi décrire l'implémentation la plus efficace de chaque algorithme, qu'il calcule ou non un arbre couvrant minimum.

- a. PEUT-ÊTRE-ACM-A(G, w)
- 1 trier les arêtes en ordre décroissant de poids d'arête w
 - 2 $T \leftarrow A$
 - 3 **pour** chaque arête a , prise par ordre décroissant de poids
 - 4 **faire si** $T - \{a\}$ est un graphe connexe
 - 5 **alors** $T \leftarrow T - a$
 - 6 **retourner** T

- b.** PEUT-ÊTRE-ACM-B(G, w)
- 1 $T \leftarrow \emptyset$
 - 2 **pour** chaque arête a prise dans un ordre arbitraire
 - 3 **faire** si $T \cup \{a\}$ n'a pas de cycles
 - 4 **alors** $T \leftarrow T \cup a$
 - 5 **retourner** T
- c.** PEUT-ÊTRE-ACM-C(G, w)
- 1 $T \leftarrow \emptyset$
 - 2 **pour** chaque arête a prise dans un ordre arbitraire
 - 3 **faire** $T \leftarrow T \cup \{a\}$
 - 4 **si** T a un cycle c
 - 5 **alors** soit a' une arête de poids maximal de c
 - 6 $T \leftarrow T - \{a'\}$
 - 7 **retourner** T

NOTES

Tarjan [292] traite le problème de l'arbre couvrant minimum et fournit une excellente étude détaillée. Un historique du problème de l'arbre couvrant minimum a été écrit par Graham et Hell [131].

Tarjan attribue le premier algorithme d'arbre couvrant minimum à un article de O. Borůvka datant de 1926. L'algorithme de Borůvka consiste à exécuter $O(\lg S)$ itérations de la procédure MST-REDUCE décrite dans le problème 23.2. L'algorithme de Kruskal a été publié par Kruskal [195] en 1956. L'algorithme connu généralement sous le nom d'algorithme de Prim fut effectivement inventé par Prim [250], mais aussi par V. Jarník antérieurement (1930).

La raison de l'efficacité des algorithmes gloutons lors de la recherche d'arbres couvrants minimum tient au fait que l'ensemble des forêts d'un graphe forme un matroïde graphique. (Voir section 16.4.)

Quand $|A| = \Omega(S \lg S)$, l'algorithme de Prim à base de tas de Fibonacci s'exécute en temps $O(A)$. Pour les graphes peu denses, en combinant les concepts des algorithmes de Prim, Kruskal et Borůvka, ainsi que des structures de données avancées, Fredman et Tarjan [98] donnent un algorithme qui s'exécute en temps $O(A \lg^* S)$. Gabow, Galil, Spencer et Tarjan [102] ont amélioré cet algorithme pour le faire s'exécuter en temps $O(A \lg \lg^* S)$. Chazelle [53] donne un algorithme qui s'exécute en temps $O(A \hat{\alpha}(A, S))$, où $\hat{\alpha}(A, S)$ est l'inverse fonctionnel de la fonction de Ackermann. (Voir notes du chapitre 21 pour une brève étude de la fonction de Ackermann et de son inverse.) Contrairement aux algorithmes antérieurs d'arbre couvrant minimum, celui de Chazelle n'est pas du genre glouton.

Un problème apparenté est la *vérification d'arbre couvrant*, où l'on a un graphe $G = (S, A)$ et un arbre $T \subseteq E$ et où l'on veut savoir si T est un arbre couvrant minimum de G . King [177] donne un algorithme à temps linéaire pour la vérification d'arbre couvrant, en s'appuyant sur des travaux antérieurs de Komlós [188] et Dixon, Rauch et Tarjan [77].

Les algorithmes susmentionnés sont tous déterministes et tombent dans le modèle basé sur les comparaisons, décrit au chapitre 8. Karger, Klein et Tarjan [169] donnent un algorithme randomisé d'arbre couvrant minimum qui s'exécute en temps moyen $O(S+A)$. Cet algorithme emploie la récursivité d'une manière semblable à celle de l'algorithme de sélection en temps linéaire vu à la section 9.3 : un appel récursif à un problème auxiliaire identifie un sous-ensemble des arêtes A' qui ne peut pas être dans un arbre couvrant minimum. Un autre appel récursif à $A - A'$ trouve alors l'arbre couvrant minimum. L'algorithme utilise aussi des idées de l'algorithme de Borůvka et de l'algorithme de King concernant la vérification d'arbre couvrant.

Fredman et Willard [100] ont montré comment trouver un arbre couvrant minimum en temps $O(S+A)$, en employant un algorithme déterministe qui ne s'appuie pas sur des comparaisons. Leur algorithme suppose que les données sont des entiers stockés sur b bits et que la mémoire de l'ordinateur est faite de mots adressables de b bits.

Chapitre 24

Plus courts chemins à origine unique

Un automobiliste souhaite trouver le plus court chemin possible entre Strasbourg et Bordeaux. Étant donnée une carte routière de France, avec les distances de chaque portion de route, comment peut-il déterminer la route la plus courte ?

Un possibilité consiste à énumérer toutes les routes de Strasbourg à Bordeaux, additionner les distances de chacune puis choisir la plus courte. Toutefois, on s'aperçoit rapidement que, même en n'autorisant pas les routes qui contiennent des cycles, il existe des millions de possibilités, dont la plupart ne valent même pas la peine d'être considérées. Par exemple, une route allant de Strasbourg à Bordeaux en passant par Lille est manifestement une mauvaise option, Lille faisant faire un détour de plusieurs centaines de kilomètres.

Dans ce chapitre, ainsi que dans le chapitre 25, nous allons montrer comment résoudre efficacement ce type de problèmes. Dans un *problème de plus courts chemins*, on possède en entrée un graphe orienté pondéré $G = (S, A)$, avec une fonction de pondération $w : A \rightarrow \mathbf{R}$ qui fait correspondre à chaque arc un poids à valeur réelle. La *longueur* du chemin $p = \langle v_0, v_1, \dots, v_k \rangle$ est la somme des poids (longueurs) des arcs qui le constituent :

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i) .$$

On définit la *longueur du plus court chemin* entre u et v par

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{s'il existe un chemin de } u \text{ à } v, \\ \infty & \text{sinon .} \end{cases}$$

Un **plus court chemin** d'un sommet u à un sommet v est alors défini comme un chemin p de longueur $w(p) = \delta(u, v)$.

Dans l'exemple Strasbourg-Bordeaux, on peut modéliser la carte routière par un graphe : les sommets représentent les intersections de route, les arcs les portions de route, et les poids les distances. Notre but est de trouver un plus court chemin entre un intersection donnée à Strasbourg (par exemple, le carrefour rue Dante/rue Wagner) et une intersection donnée à Bordeaux (par exemple, le carrefour rue Montaigne/rue La Boétie).

Les poids des arcs peuvent mesurer autre chose que des distances. On s'en sert souvent pour représenter un temps, un coût, des pénalités, une déperdition, ou n'importe quelle autre quantité qui s'accumule linéairement le long d'un chemin, et qu'il s'agit de minimiser.

L'algorithme de parcours en largeur de la section 22.2 est un algorithme de recherche des plus courts chemins, qui fonctionne sur des graphes non pondérés, c'est-à-dire des graphes pour lesquels on considère que chaque arc possède un poids unitaire. Bon nombre des concepts du parcours en largeur se retrouvent dans l'étude des plus courts chemins dans les graphes pondérés. Le lecteur est donc encouragé à relire la section 22.2 avant de continuer.

a) Variantes

Dans ce chapitre, nous restreindrons notre étude au **problème de recherche du plus court chemin à origine unique** : étant donné un graphe $G = (S, A)$, on souhaite trouver un plus court chemin depuis un sommet **origine** donné $s \in S$ vers n'importe quel sommet $v \in S$. Beaucoup d'autres problèmes peuvent être résolus par l'algorithme à origine unique, notamment les variantes suivantes :

Plus court chemin à destination unique : Trouver un plus court chemin vers un sommet de **destination** t à partir de n'importe quel sommet v . En inversant le sens de chaque arc du graphe, on peut ramener ce problème à un problème à origine unique.

Plus court chemin pour un couple de sommets donné : Trouver un plus court chemin de u à v pour deux sommets donnés u et v . Si on résout le problème à origine unique pour le sommet origine u , on résout ce problème également. Par ailleurs, on ne connaît aucun algorithme qui soit meilleur asymptotiquement que les meilleurs algorithmes à origine unique dans le pire des cas.

Plus court chemin pour tout couple de sommets : Trouver un plus court chemin de u à v pour tout couple de sommets u et v . Ce problème peut être résolu en exécutant un algorithme à origine unique à partir de chaque sommet ; mais on peut généralement le résoudre plus rapidement, et sa structure est intéressante en elle-même. Le chapitre 25 traitera en détail ce problème.

b) Sous-structure optimale d'un plus court chemin

Les algorithmes de plus court chemin s'appuient généralement sur la propriété selon laquelle un plus court chemin entre deux sommets contient d'autres plus courts chemins. (L'algorithme de flot maximal d'Edmonds-Karp du chapitre 26 repose également sur cette propriété.) Cette propriété de sous-structure optimale est une caractéristique de l'applicabilité tant de la programmation dynamique (chapitre 15) que des méthodes gloutonnes (chapitre 16). L'algorithme de Dijkstra, que nous verrons à la section 24.3, est un algorithme glouton et l'algorithme de Floyd-Warshall, qui trouve les plus courts chemins entre toutes les paires de sommets (voir chapitre 25), est un algorithme de programmation dynamique. Le lemme suivant énonce la propriété de sous-structure optimale des plus courts chemins d'une manière plus formelle.

Lemme 24.1 (les sous-chemins de plus courts chemins sont des plus courts chemins) *Étant donné un graphe orienté pondéré $G = (S, A)$ ayant la fonction de pondération $w : A \rightarrow \mathbf{R}$, soit $p = \langle v_1, v_2, \dots, v_k \rangle$ un plus court chemin du sommet v_1 au sommet v_k et, pour tout i et tout j tels que $1 \leq i \leq j \leq k$, soit $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ le sous-chemin de p entre le sommet v_i et le sommet v_j . Alors, p_{ij} est un plus court chemin de v_i à v_j .*

Démonstration : Si l'on décompose le chemin p en $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, on a alors $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Supposons maintenant qu'il y ait un chemin p'_{ij} entre v_i et v_j de longueur $w(p'_{ij}) < w(p_{ij})$. Alors, $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ est un chemin de v_1 à v_k dont la longueur $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ est inférieure à $w(p)$, ce qui contredit l'hypothèse selon laquelle p est un plus court chemin de v_1 à v_k . \square

c) Arcs de poids négatif

Pour certaines instances du problème du plus court chemin à origine unique, on peut rencontrer des arcs dont les poids sont négatifs. Si le graphe $G = (S, A)$ ne contient aucun circuit de longueur strictement négative accessible à partir de l'origine s , alors, pour tout $v \in S$, la longueur du plus court chemin $\delta(s, v)$ reste bien défini, même si sa valeur est négative. Toutefois, s'il existe un circuit de longueur strictement négative accessible depuis s , la longueur d'un plus court chemin n'est plus bien défini. Aucun chemin entre s et un sommet du circuit ne peut être un plus court chemin : on peut toujours trouver un chemin de moindre longueur qui suit le « plus court » chemin puis traverse le circuit de longueur strictement négative. S'il existe un circuit de longueur strictement négative sur un chemin de s à v , on définit $\delta(s, v) = -\infty$.

La figure 24.1 illustre l'effet des poids négatifs et des circuits de longueur strictement négative sur les longueurs des plus courts chemins. Comme il n'existe qu'un seul chemin de s vers a (le chemin $\langle s, a \rangle$), $\delta(s, a) = w(s, a) = 3$. De même, il n'existe qu'un seul chemin de s vers b , et $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Il existe une infinité de chemins de s vers c : $\langle s, c \rangle$, $\langle s, c, d, c \rangle$, $\langle s, c, d, c, d, c \rangle$, etc. Le circuit $\langle c, d, c \rangle$ ayant une longueur $6 + (-3) = 3 > 0$, le plus court chemin de s vers c est

$\langle s, c \rangle$, de poids $\delta(s, c) = 5$. De même, le plus court chemin de s vers d est $\langle s, c, d \rangle$, pour une longueur $\delta(s, d) = w(s, c) + w(c, d) = 11$. De la même manière, on peut trouver une infinité de chemins de s vers e : $\langle s, e \rangle$, $\langle s, e, f, e \rangle$, $\langle s, e, f, e, f, e \rangle$, etc. Cependant le circuit $\langle e, f, e \rangle$ ayant pour longueur $3 + (-6) = -3 < 0$, il n'existe pas de plus court chemin de s vers e . En traversant un nombre de fois arbitraire le circuit de longueur strictement négative $\langle e, f, e \rangle$, on peut trouver des chemins entre s et e avec des longueurs strictement négatives arbitrairement grandes en valeur absolue ; et donc $\delta(s, e) = -\infty$. De même, $\delta(s, f) = -\infty$. Comme g est accessible à partir de f , on peut aussi trouver des chemins ayant des longueurs strictement négatives arbitrairement grandes en valeur absolue entre s et g , et $\delta(s, g) = -\infty$. Les sommets h, i , et j forment également un circuit de longueur strictement négative. Toutefois, ils ne sont pas accessibles à partir de s , et donc $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

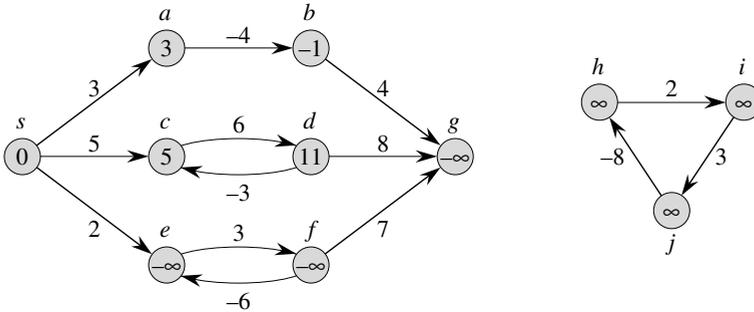


Figure 24.1 Pondérations négatives dans un graphe orienté. À chaque sommet est associé le poids de plus court chemin entre l'origine s et ce sommet. Comme les sommets e et f appartiennent à un circuit de longueur strictement négative accessible à partir de s , leur poids associé vaut $-\infty$. Puisque le sommet g est accessible à partir d'un sommet dont la longueur de plus court chemin est $-\infty$, son poids associé vaut également $-\infty$. Les sommets h, i , et j n'étant pas accessibles à partir de s , leur poids associé vaut ∞ , bien qu'ils se trouvent à l'intérieur d'un circuit de longueur strictement négative.

Certains algorithmes de recherche du plus court chemin, tel l'algorithme de Dijkstra, supposent que tous les poids d'arc du graphe d'entrée sont positifs ou nuls, comme dans l'exemple de la carte routière. D'autres, comme celui de Bellman-Ford autorisent des arcs de poids négatifs dans le graphe d'entrée, et fournissent une réponse correcte, du moment qu'aucun circuit de longueur strictement négative n'est accessible à partir de l'origine. En général, si un tel circuit existe, l'algorithme peut détecter et signaler son existence.

d) Circuit

Un plus court chemin peut-il contenir un circuit ? Nous venons de voir qu'il ne peut pas contenir de circuit de longueur strictement négative. Il ne peut pas, non plus, contenir de circuit de longueur strictement positive ; en effet, en supprimant le circuit du chemin, on obtient un chemin de même origine, de même destination et de longueur strictement inférieure. Autrement dit, si $p = \langle v_0, v_1, \dots, v_k \rangle$ est un chemin

et $c = \langle v_i, v_{i+1}, \dots, v_j \rangle$ un circuit de longueur strictement positive sur ce chemin (de sorte que $v_i = v_j$ et $w(c) > 0$), alors le chemin $p' = \langle v_0, v_1, \dots, v_i, v_{j+1}, v_{j+2}, \dots, v_k \rangle$ a une longueur $w(p') = w(p) - w(c) < w(p)$, et donc p ne peut pas être un plus court chemin entre v_0 et v_k .

Cela ne laisse donc que les circuits de poids nul. On peut supprimer un circuit de poids nul d'un chemin pour obtenir un autre chemin de même poids. Donc, s'il existe un plus court chemin entre un sommet origine s et un sommet destination v qui contient un circuit de poids nul, alors il existe un autre plus court chemin entre s et v privé de ce circuit. Tant qu'un plus court chemin a des circuits de poids nul, on peut les supprimer l'un après l'autre jusqu'à obtenir un plus court chemin débarrassé de tout circuit. Par conséquent, on peut supposer sans nuire à la généralité que, quand nous trouverons des plus courts chemins, ils n'auront pas de circuits. Comme un chemin élémentaire d'un graphe $G = (S, A)$ contient au plus $|S|$ sommets distincts, il contient au plus $|S| - 1$ arcs. Donc, on pourra limiter notre étude aux plus courts chemins ayant au plus $|S| - 1$ arcs.

e) Représentation des plus courts chemins

On souhaite souvent calculer non seulement les poids des plus courts chemins, mais aussi les sommets présents sur ces plus courts chemins. La représentation utilisée pour les plus courts chemins ressemble à celle utilisée pour les arborescences de parcours en largeur de la section 22.2. Étant donné un graphe $G = (S, A)$, on gère pour chaque sommet $v \in S$ un **prédécesseur** $\pi[v]$ qui est soit un autre sommet, soit NIL. Les algorithmes de recherche du plus court chemin exposés dans ce chapitre gèrent les attributs π de manière que la chaîne des prédécesseurs partant d'un sommet v suive, en remontant, un plus court chemin entre s et v . Ainsi, étant donné un sommet v pour lequel $\pi[v] \neq \text{NIL}$, la procédure IMPRIMER-CHEMIN(G, s, v) de la section 22.2 peut servir à imprimer un plus court chemin de s vers v .

Pendant, lors de l'exécution d'un algorithme de recherche du plus court chemin, les valeurs π n'ont pas besoin d'indiquer les plus courts chemins. Comme dans le parcours en largeur, nous nous intéresserons au **sous-graphe prédécesseur** $G_\pi = (S_\pi, A_\pi)$ induit par les valeurs π . Ici aussi, on définira l'ensemble S_π comme étant l'ensemble des sommets de G ayant des prédécesseurs différents de NIL, complété par l'origine s :

$$S_\pi = \{v \in S : \pi[v] \neq \text{NIL}\} \cup \{s\} .$$

L'ensemble A_π est l'ensemble des arcs induits par les valeurs de π pour les sommets appartenant à S_π :

$$A_\pi = \{(\pi[v], v) \in A : v \in S_\pi - \{s\}\} .$$

Nous allons prouver que les valeurs de π obtenues par les algorithmes de ce chapitre ont pour propriété que, lorsqu'ils se terminent, G_π est une « arborescence de plus courts chemins » ; de manière informelle, disons que c'est une arborescence contenant des plus courts chemins allant de l'origine s vers tous les sommets accessibles

depuis s . Une arborescence de plus courts chemins ressemble aux arborescences de parcours en largeur vus à la section 22.2, mais les plus courts chemins à partir de son origine sont exprimés non pas en fonction du nombre d'arcs, mais du poids des arcs. Plus précisément, soit $G = (S, A)$ un graphe orienté pondéré, et une fonction de pondération associée $w : A \rightarrow \mathbf{R}$. On suppose que G ne contient aucun circuit de longueur strictement négative accessible depuis le sommet origine $s \in S$, de sorte que les plus courts chemins sont bien définis. Une **arborescence de plus courts chemins** de racine s est un sous-graphe orienté $G' = (S', A')$, où $S' \subseteq S$ et $A' \subseteq A$, tel que

- 1) S' est l'ensemble des sommets accessibles à partir de s dans G ,
- 2) G' forme une arborescence de racine s , et
- 3) pour tout $v \in S'$, le chemin unique de s à v dans G' est un plus court chemin de s vers v dans G .

Les plus courts chemins ne sont pas forcément uniques, pas plus que les arborescences de plus courts chemins. Par exemple, la figure 24.2 montre un graphe orienté pondéré et deux arborescences de plus courts chemins ayant la même racine.

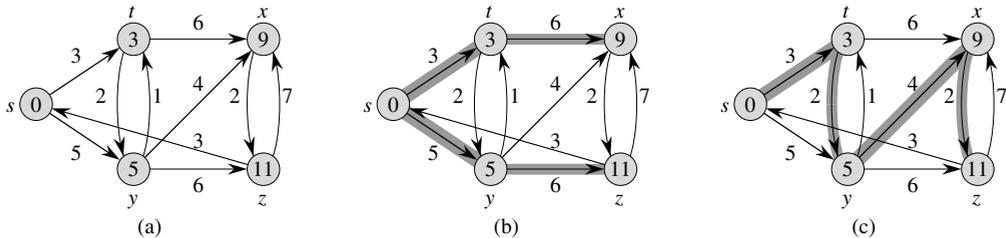


Figure 24.2 (a) Un graphe orienté pondéré avec des poids des plus courts chemins à partir de l'origine s . (b) Les arcs en gris forment une arborescence de plus courts chemins de racine s . (c) Une autre arborescence de plus courts chemins de même racine.

f) Relâchement

Les algorithmes de ce chapitre emploient la technique du **relâchement**. Pour chaque sommet $v \in S$, on gère un attribut $d[v]$ qui est un majorant de la longueur d'un plus court chemin entre l'origine s et v . On appelle $d[v]$ une **estimation de plus court chemin**. On initialise les estimations et les prédécesseurs via la procédure en temps $\Theta(S)$ que voici.

SOURCE-UNIQUE-INITIALISATION(G, s)

- 1 **pour** chaque sommet $v \in S[G]$
- 2 **faire** $d[v] \leftarrow \infty$
- 3 $\pi[v] \leftarrow \text{NIL}$
- 4 $d[s] \leftarrow 0$

Après initialisation, $\pi[v] = \text{NIL}$ pour tout $v \in S$, $d[s] = 0$ et $d[v] = \infty$ pour $v \in S - \{s\}$.

Le processus de **relâchement** ⁽¹⁾ d'un arc (u, v) consiste à tester si l'on peut améliorer le plus court chemin vers v trouvé jusqu'ici en passant par u et, si tel est le cas, en actualisant $d[v]$ et $\pi[v]$. Une étape de relâchement peut diminuer la valeur de l'estimation de plus court chemin $d[v]$ et mettre à jour le champ prédécesseur $\pi[v]$ de v . Le code suivant effectue une étape de relâchement sur l'arc (u, v) .

```
RELÂCHER( $u, v, w$ )
1  si  $d[v] > d[u] + w(u, v)$ 
2    alors  $d[v] \leftarrow d[u] + w(u, v)$ 
3     $\pi[v] \leftarrow u$ 
```

La figure 24.3 montre deux exemples de relâchement d'arc, un dans lequel l'estimation de plus court chemin diminue et un autre dans lequel il n'y a pas modification de l'estimation.

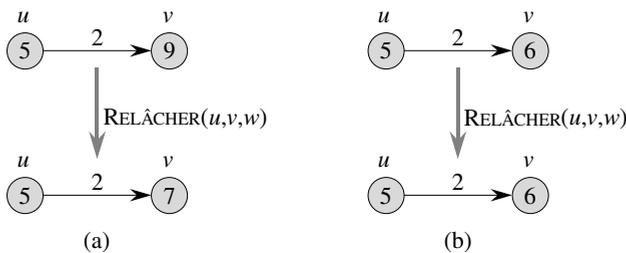


Figure 24.3 Relâchement d'un arc (u, v) de poids $w(u, v) = 2$. L'estimation de plus court chemin de chaque sommet est affichée dans le sommet. (a) Comme $d[v] > d[u] + w(u, v)$ avant relâchement, la valeur de $d[v]$ décroît. (b) Ici, $d[v] \leq d[u] + w(u, v)$ avant l'étape de relâchement, de sorte que $d[v]$ ne change pas avec le relâchement.

Chaque algorithme de ce chapitre appelle SOURCE-UNIQUE-INITIALISATION, puis relâche les arcs de manière réitérée. En outre, le relâchement est la seule façon de modifier les estimations de plus court chemin et les prédécesseurs. Les algorithmes de ce chapitre diffèrent par le nombre de fois qu'ils relâchent chaque arc et par l'ordre dans lequel ils relâchent les arcs. Dans l'algorithme de Dijkstra et dans l'algorithme de plus court chemin pour graphe sans circuit orienté, chaque arc est relâché une fois et une seule. Dans l'algorithme de Bellman-Ford, chaque arc est relâché plusieurs fois.

g) Propriétés de plus courts chemins et relâchement

Pour prouver la conformité des algorithmes de ce chapitre, nous ferons appel à plusieurs propriétés des plus courts chemins et au relâchement. Nous énoncerons ces

(1) Il peut paraître étrange que le terme « relâchement » désigne une opération qui affine un majorant. Cela tient à des raisons historiques. Le résultat d'une étape de relâchement peut être vu comme un relâchement de la contrainte $d[v] \leq d[u] + w(u, v)$, ce qui, d'après l'inégalité triangulaire (lemme 24.10), est forcément vrai si $d[u] = \delta(s, u)$ et $d[v] = \delta(s, v)$. En d'autres termes, si $d[v] \leq d[u] + w(u, v)$, il n'y a pas de « pression » pour satisfaire à cette contrainte, de sorte que la contrainte est « relâchée ».

propriétés ici, puis la section 24.5 les démontrera formellement. Pour que vous puissiez trouver les références, chaque propriété énoncée ici inclut le numéro de lemme ou de corollaire idoine provenant de la section 24.5. Les cinq dernières de ces propriétés, qui concernent les estimations de plus court chemin ou le sous-graphe prédécesseur, supposent implicitement que le graphe est initialisé par un appel à SOURCE-UNIQUE-INITIALISATION(G, s) et que l'unique façon de modifier les estimations de plus court chemin et le sous-graphe prédécesseur passe par une certaine suite d'étapes de relâchement.

Inégalité triangulaire : (lemme 24.10)

Pour tout arc $(u, v) \in A$, on a $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Propriété du majorant : (lemme 24.11)

On a toujours $d[v] \geq \delta(s, v)$ pour tous les sommets $v \in S$, et une fois que $d[v]$ a atteint la valeur $\delta(s, v)$, elle ne change plus.

Propriété aucun-chemin : (corollaire 24.12)

S'il n'y a pas de chemin de s à v , alors on a toujours $d[v] = \delta(s, v) = \infty$.

Propriété de convergence : (lemme 24.14)

Si $s \rightsquigarrow u \rightarrow v$ est un plus court chemin dans G pour un certain $u, v \in S$ et si $d[u] = \delta(s, u)$ à un certain instant antérieur au relâchement de l'arc (u, v) , alors $d[v] = \delta(s, v)$ en permanence après le relâchement.

Propriété de relâchement de chemin : (lemme 24.15)

Si $p = \langle v_0, v_1, \dots, v_k \rangle$ est un plus court chemin de $s = v_0$ à v_k et si les arcs de p sont relâchés dans l'ordre $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, alors $d[v_k] = \delta(s, v_k)$. Cette propriété est vraie indépendamment de toutes autres étapes de relâchement susceptibles de se produire, même si elles s'entremêlent avec des relâchements d'arcs de p .

Propriété de sous-graphe prédécesseur : (lemme 24.17)

Une fois que $d[v] = \delta(s, v)$ pour tout $v \in S$, le sous-graphe prédécesseur est une arborescence de plus courts chemins de racine s .

h) Structure du chapitre

La section 24.1 présentera l'algorithme de Bellman-Ford, qui résout le problème des plus courts chemins à origine unique dans le cas général où les arcs peuvent avoir des poids négatifs. L'algorithme de Bellman-Ford est remarquable de simplicité, et il présente l'avantage supplémentaire de détecter si un circuit de longueur strictement négative est accessible depuis l'origine. La section 24.2 donnera un algorithme à temps linéaire pour calculer les plus courts chemins depuis une origine unique dans un graphe orienté sans circuit. La section 24.3 couvrira l'algorithme de Dijkstra, qui a un temps d'exécution inférieur à l'algorithme de Bellman-Ford mais qui exige que les arcs aient des poids positifs. La section 24.4 montrera comment l'algorithme de Bellman-Ford permet de résoudre un cas particulier de « programmation linéaire ». Enfin, la section 24.5 démontrera les propriétés de plus court chemin et de relâchement susmentionnées.

Notre analyse reposera sur quelques conventions permettant d'effectuer des opérations arithmétiques avec les infinis. On supposera que, pour un nombre réel $a \neq -\infty$ quelconque, on a $a + (+\infty) = (+\infty) + a = +\infty$. D'autre part, pour que les démonstrations restent valables en présence de circuits de longueur strictement négative, on fera l'hypothèse que, pour un nombre réel $a \neq +\infty$ quelconque, $a + (-\infty) = (-\infty) + a = -\infty$.

Tous les algorithmes de ce chapitre supposent que le graphe orienté G est représenté par des listes d'adjacences. En outre, le poids d'un arc est stocké avec l'arc ; ainsi, quand on parcourt chaque liste d'adjacences, on peut déterminer les poids d'arc avec un temps de $O(1)$ par arc.

24.1 ALGORITHME DE BELLMAN-FORD

L'algorithme de Bellman-Ford résout le problème des plus courts chemins à origine unique dans le cas général où les poids d'arc peuvent avoir des valeurs négatives. Étant donné un graphe orienté pondéré $G = (S, A)$, de fonction de pondération $w : A \rightarrow \mathbf{R}$, et une origine s , l'algorithme de Bellman-Ford retourne une valeur booléenne indiquant s'il existe ou non un circuit de longueur strictement négative accessible à partir de s . Si un tel circuit n'existe pas, l'algorithme donne les plus courts chemins ainsi que leurs poids.

L'algorithme utilise la technique du relâchement, diminuant progressivement une estimation $d[v]$ du poids d'un plus court chemin depuis l'origine s vers chaque sommet $v \in S$ jusqu'à atteindre la valeur réelle du poids de plus court chemin, $\delta(s, v)$. L'algorithme retourne VRAI si et seulement si le graphe ne contient aucun circuit de longueur strictement négative accessible depuis l'origine.

```

BELLMAN-FORD( $G, w, s$ )
1 SOURCE-UNIQUE-INITIALISATION( $G, s$ )
2 pour  $i \leftarrow 1$  à  $|S[G]| - 1$ 
3   faire pour chaque arc  $(u, v) \in A[G]$ 
4     faire RELÂCHER( $u, v, w$ )
5 pour chaque arc  $(u, v) \in A[G]$ 
6   faire si  $d[v] > d[u] + w(u, v)$ 
7     alors retourner FAUX
8 retourner VRAI

```

La figure 24.4 montre comment l'exécution de l'algorithme de Bellman-Ford fonctionne sur un graphe de 5 sommets. Après l'initialisation des valeurs d et π de tous les sommets en ligne 1, l'algorithme effectue $|S| - 1$ passages sur les arcs du graphe. Chaque passage correspond à une itération de la boucle **pour** des lignes 2–4 et consiste à relâcher chaque arc du graphe une seule fois. Les figures 24.4(b)–(e) montrent l'état de l'algorithme après chacun des quatre passages. Après les $|S| - 1$

passages, les lignes 5–8 testent la présence d'un circuit de longueur strictement négative et retournent la valeur booléenne appropriée. (Nous justifierons un peu plus tard le fonctionnement de ce test).

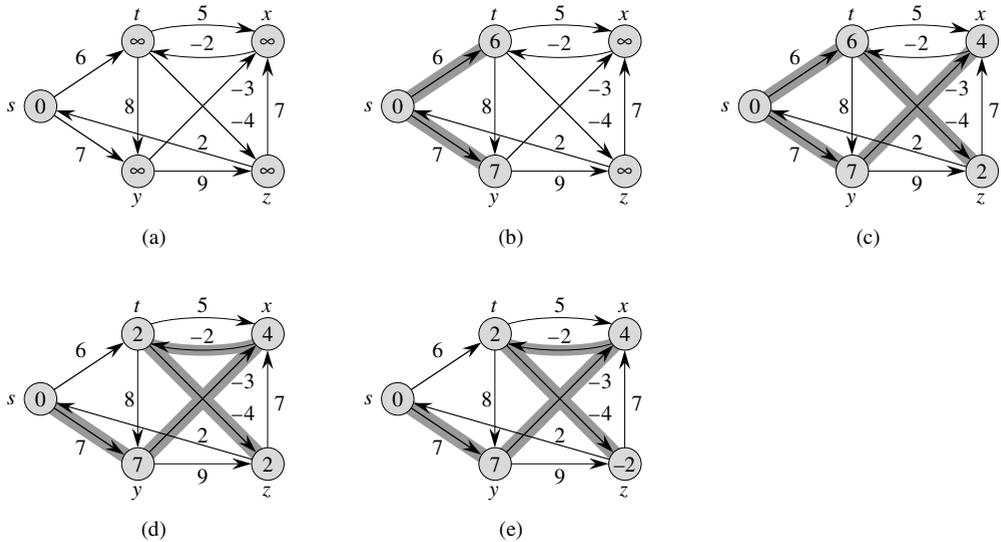


Figure 24.4 L'exécution de l'algorithme de Bellman-Ford. Le sommet origine est s . Les valeurs de d sont représentées dans les sommets, et les arcs en gris indiquent les valeurs prédécesseur : si l'arc (u, v) est en gris, alors $\pi[v] = u$. Dans cet exemple particulier, chaque passage relâche les arcs dans l'ordre (t, x) , (t, y) , (t, z) , (x, t) , (y, x) , (y, z) , (z, x) , (z, s) , (s, t) , (s, y) . (a) La situation juste avant le premier passage sur les arcs. (b)–(e) La situation après chacun des passages suivants. Les valeurs d et π donnée en partie (e) sont les valeurs finales. L'algorithme de Bellman-Ford retourne VRAI pour cet exemple.

L'algorithme de Bellman-Ford s'exécute en temps $O(SA)$: l'initialisation de la ligne 1 prend $\Theta(S)$; chacun des $|S| - 1$ passages des lignes 2–4 prend $\Theta(A)$; la boucle **pour** des lignes 5–7 prend $O(A)$.

Pour démontrer la validité de l'algorithme de Bellman-Ford, on commence par montrer que, s'il n'existe aucun circuit de longueur strictement négative, l'algorithme calcule les longueurs de plus court chemin corrects pour tous les sommets accessibles depuis l'origine.

Lemme 24.2 Soit $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$ et d'origine s ; on suppose que G ne contient aucun circuit de longueur strictement négative accessible depuis s . Alors, après les $|S| - 1$ itérations de la boucle **pour** des lignes 2–4 de BELLMAN-FORD, on a $d[v] = \delta(s, v)$ pour tous les sommets v accessibles depuis s .

Démonstration : On démontre le lemme en faisant appel à la propriété de relâchement de chemin. Soit un sommet v accessible depuis s , et soit $p = \langle v_0, v_1, \dots, v_k \rangle$, où $v_0 = s$ et $v_k = v$, un plus court chemin élémentaire de s à v . Le chemin p a au plus

$|S| - 1$ arcs, et donc $k \leq |S| - 1$. Chacune des $|S| - 1$ itérations de la boucle **pour** des lignes 2–4 relâche tous les $|A|$ arcs. Parmi les arcs relâchés dans la i ème itération, pour $i = 1, 2, \dots, k$, il y a (v_{i-1}, v_i) . D'après la propriété de relâchement de chemin, on a donc $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. \square

Corollaire 24.3 Soit $G = (S, A)$ un graphe orienté pondéré de fonction de pondération $w : A \rightarrow \mathbf{R}$ et d'origine s . Alors, pour chaque sommet $v \in S$, il existe un chemin de s vers v si et seulement si BELLMAN-FORD se termine avec $d[v] < \infty$ quand elle est exécutée sur G .

Démonstration : La démonstration est laissée à l'exercice 24.1.2. \square

Théorème 24.4 (Validité de l'algorithme de Bellman-Ford) Exécutons la procédure BELLMAN-FORD sur un graphe orienté pondéré $G = (S, A)$, de fonction de pondération $w : A \rightarrow \mathbf{R}$ et d'origine s . Si G ne contient aucun circuit de longueur strictement négative accessible depuis s , alors l'algorithme retourne VRAI, on a $d[v] = \delta(s, v)$ pour tous les sommets $v \in S$, et le sous-graphe prédécesseur G_π est une arborescence de plus courts chemins de racine s . Si G contient un circuit de longueur strictement négative accessible à partir de s , alors l'algorithme retourne FAUX.

Démonstration : Supposons que le graphe G ne contienne aucun circuit de longueur strictement négative accessible à partir de l'origine s . Nous commençons par démontrer l'assertion selon laquelle, à la fin de l'algorithme, $d[v] = \delta(s, v)$ pour tous les sommets $v \in S$. Si le sommet v est accessible depuis s , le lemme 24.2 démontre cette assertion. Si v n'est pas accessible à partir de s , l'assertion se déduit de la propriété aucun-chemin. Elle est donc vérifiée. La propriété de sous-graphe prédécesseur, combinée avec l'assertion, implique que G_π est une arborescence de plus courts chemins. Utilisons maintenant l'assertion pour montrer que BELLMAN-FORD retourne VRAI. Après exécution de l'algorithme, on a pour tout arc $(u, v) \in A$,

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) && \text{(d'après l'inégalité triangulaire)} \\ &= d[u] + w(u, v), \end{aligned}$$

et donc aucun des tests de la ligne 6 ne force BELLMAN-FORD à retourner FAUX. Il retourne donc VRAI.

Inversement, supposons que le graphe G contienne un circuit de longueur strictement négative $c = \langle v_0, v_1, \dots, v_k \rangle$, où $v_0 = v_k$, accessible depuis l'origine s . Alors,

$$\sum_{i=1}^k w(v_{i-1}, v_i) < 0. \quad (24.1)$$

Raisonnons par l'absurde en supposant que l'algorithme de Bellman-Ford retourne VRAI. Alors, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ pour $i = 1, 2, \dots, k$. La sommation des

inégalités sur le circuit c donne

$$\begin{aligned} \sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i). \end{aligned}$$

Comme $v_0 = v_k$, chaque sommet de c apparaît exactement une fois dans chacune des sommes. Donc

$$\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}].$$

De plus, d'après le corollaire 24.3, $d[v_i]$ est fini pour $i = 1, 2, \dots, k$. Donc

$$0 \leq \sum_{i=1}^k w(v_{i-1}, v_i),$$

ce qui contredit l'inégalité (24.1). On en conclut que l'algorithme de Bellman-Ford retourne VRAI si le graphe G ne contient aucun circuit de longueur strictement négative accessible depuis l'origine, et FAUX sinon. \square

Exercices

24.1.1 Exécuter l'algorithme de Bellman-Ford sur le graphe orienté de la figure 24.4, en prenant z pour origine. À chaque passage, relâcher les arcs dans le même ordre que sur la figure, et donner les valeurs de d et π après chaque passage. Ensuite, donner au poids de l'arc (z, x) la valeur 4, et exécuter à nouveau l'algorithme en prenant pour origine le sommet s .

24.1.2 Démontrer le corollaire 24.3.

24.1.3 Étant donné un graphe orienté pondéré $G = (S, A)$ sans circuit de longueur strictement négative, soit m le maximum, pour tous les couples de sommets $u, v \in S$, du nombre minimal d'arcs dans un plus court chemin de u à v . (Ici, « plus court » signifie de poids minimal et ne concerne pas le nombre d'arcs). Suggérer une modification simple à l'algorithme de Bellman-Ford, lui permettant de se terminer après $m + 1$ passages.

24.1.4 Modifier l'algorithme de Bellman-Ford pour qu'il donne à $d[v]$ la valeur $-\infty$ pour tous les sommets v pour lesquels il existe un circuit de longueur strictement négative sur un certain chemin entre l'origine et v .

24.1.5 Soit $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$. Donner un algorithme à temps $O(SA)$ qui détermine, pour chaque sommet $v \in S$, la valeur $\delta^*(v) = \min_{u \in S} \{\delta(u, v)\}$.

24.1.6 * Supposons qu'un graphe orienté pondéré $G = (S, A)$ comporte un circuit de longueur strictement négative. Donner un algorithme efficace permettant de lister les sommets d'un tel circuit. Démontrer la validité de votre algorithme.

24.2 PLUS COURTS CHEMINS À ORIGINE UNIQUE DANS LES GRAPHES ORIENTÉS SANS CIRCUIT

En relâchant les arcs d'un graphe orienté acyclique $G = (S, A)$ dans l'ordre topologique des sommets, on peut calculer les plus courts chemins à partir d'une origine unique en temps $\Theta(S + A)$. Les plus courts chemins sont toujours bien définis dans un graphe orienté sans circuit car, même s'il existe des arcs de poids négatif, aucun circuit de longueur strictement négative ne peut exister.

L'algorithme commence par trier topologiquement le graphe orienté sans circuit (voir section 22.4) pour imposer aux sommets un ordre linéaire. S'il existe un chemin allant du sommet u au sommet v , alors u précède v dans l'ordre topologique. Un seul passage est effectué sur les sommets triés topologiquement. Pendant le traitement de chaque sommet, tous les arcs qui partent du sommet sont relâchés.

PLUS-COURTS-CHEMINS-GSS(G, w, s)

- 1 trier topologiquement les sommets de G
- 2 SOURCE-UNIQUE-INITIALISATION(G, s)
- 3 **pour** chaque sommet u pris dans l'ordre topologique
- 4 **faire pour** chaque sommet $v \in Adj[u]$
- 5 **faire** RELÂCHER(u, v, w)

On peut voir un exemple d'exécution de cet algorithme à la figure 24.5.

Le temps d'exécution de cet algorithme est facile à analyser. Comme on l'a vu à la section 22.4, le tri topologique de la ligne 1 peut s'effectuer en temps $\Theta(S + A)$. L'appel à SOURCE-UNIQUE-INITIALISATION en ligne 2 prend dans un temps $\Theta(S)$. Dans la boucle **pour** des lignes 3–5, il y a une itération par sommet. Pour chaque sommet, les arcs qui partent du sommet sont examinés un par un exactement une fois. Il y a donc en tout $|A|$ itérations de la boucle **pour** intérieure des lignes 4–5. (On a employé ici une analyse par agrégat.) Comme chaque itération de la boucle **pour** intérieure prend un temps $\Theta(1)$, le temps d'exécution total est donc $\Theta(S + A)$, donc linéaire par rapport à la taille d'une représentation du graphe par listes d'adjacences.

Le théorème suivant montre que la procédure PLUS-COURTS-CHEMINS-GSS calcule correctement les plus courts chemins.

Théorème 24.5 *Si un graphe orienté pondéré $G = (S, A)$ d'origine s ne contient aucun circuit, alors après exécution de la procédure PLUS-COURTS-CHEMINS-GSS, $d[v] = \delta(s, v)$ pour tous les sommets $v \in S$, et le sous-graphe prédécesseur G_π est une arborescence de plus courts chemins.*

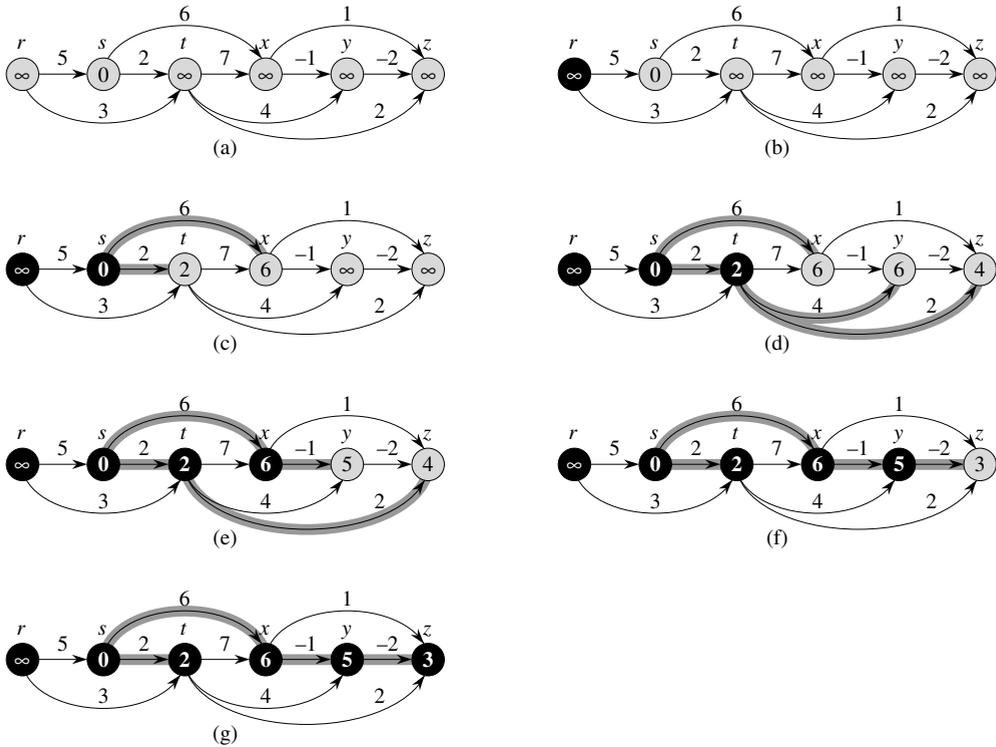


Figure 24.5 L'exécution de l'algorithme de recherche des plus courts chemins dans un graphe orienté sans circuit. Les sommets sont triés topologiquement de gauche à droite. Le sommet origine est s . Les valeurs de d sont représentées à l'intérieur des sommets, et les arcs en gris indiquent les valeurs de π . (a) La situation avant la première itération de la boucle **pour** des lignes 3–5. (b)–(g) La situation après chaque itération de la boucle **pour** des lignes 3–5. Le sommet nouvellement noirci à chaque itération est celui choisi comme sommet u de cette itération. Les valeurs montrées dans la partie (g) sont les valeurs finales.

Démonstration : On commence par montrer que $d[v] = \delta(s, v)$ pour tous les sommets $v \in S$ après l'exécution. Si v n'est pas accessible depuis s , on a $d[v] = \delta(s, v) = \infty$ d'après la propriété aucun-chemin. A présent, on suppose que v est accessible depuis s , de sorte qu'il existe un plus court chemin $p = \langle v_0, v_1, \dots, v_k \rangle$, où $v_0 = s$ et $v_k = v$. Comme les sommets sont traités dans un ordre topologique, les arcs qui composent p sont relâchés dans l'ordre $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. La propriété de relâchement de chemin implique que $d[v_i] = \delta(s, v_i)$ après l'exécution, pour $i = 0, 1, \dots, k$. Enfin, d'après la propriété de sous-graphe prédécesseur, G_π est une arborescence de plus courts chemins. \square

On retrouve une application intéressante de cet algorithme quand on détermine les chemins critiques dans l'analyse d'un **diagramme PERT** ⁽²⁾. Les arcs représentent

(2) PERT est l'acronyme de « program evaluation and review technique ».

des tâches à effectuer, et les poids représentent les temps nécessaire à l'exécution des tâches. Si l'arc (u, v) arrive à un sommet v et si l'arc (v, x) part de v , c'est que la tâche (u, v) doit être effectuée avant la tâche (v, x) . Un chemin dans le graphe orienté sans circuit représente une séquence de tâches qui doivent être effectuées dans un certain ordre. Un **chemin critique** est un *plus long* chemin, qui correspond au temps maximal requis pour effectuer une séquence ordonnée de tâches. La longueur d'un chemin critique est un minorant du temps total nécessaire à l'exécution de toutes les tâches. On peut trouver un chemin critique, soit

- en prenant l'opposé des poids d'arc et en exécutant PLUS-COURTS-CHEMINS-GSS, soit
- en exécutant PLUS-COURTS-CHEMINS-GSS, mais en remplaçant « ∞ » par « $-\infty$ » à la ligne 2 de SOURCE-UNIQUE-INITIALISATION et « $>$ » par « $<$ » dans la procédure RELÂCHER.

Exercices

24.2.1 Exécuter PLUS-COURTS-CHEMINS-GSS sur le graphe orienté de la figure 24.5, en prenant pour origine le sommet r .

24.2.2 Supposons que la ligne 3 de PLUS-COURTS-CHEMINS-GSS soit remplacée par

- 3 **pour** les $|S| - 1$ premiers sommets, pris dans l'ordre topologique

Montrer que la procédure est encore valable.

24.2.3 La formulation précédente du diagramme PERT va un peu contre l'intuition. Il serait plus naturel que les sommets représentent les tâches et que les arcs représentent les contraintes de séquençement ; autrement dit, l'arc (u, v) indiquerait qu'il faut faire la tâche u avant la tâche v . Les poids seraient alors affectés aux sommets, et non aux arcs. Modifier la procédure PLUS-COURTS-CHEMINS-GSS pour qu'elle trouve en temps linéaire un plus long chemin dans un graphe orienté sans circuit à sommets pondérés.

24.2.4 Donner un algorithme efficace permettant de compter le nombre total de chemins dans un graphe orienté sans circuit. Analyser votre algorithme.

24.3 ALGORITHME DE DIJKSTRA

L'algorithme de Dijkstra résout le problème de la recherche d'un plus court chemin à origine unique pour un graphe orienté pondéré $G = (S, A)$ dans le cas où tous les arcs ont des poids positifs ou nuls. Dans cette section, on supposera donc que $w(u, v) \geq 0$ pour chaque arc $(u, v) \in A$. Comme nous le verrons, pour peu qu'il soit bien implémenté, l'algorithme de Dijkstra a un temps d'exécution inférieur à celui de Bellman-Ford.

L'algorithme de Dijkstra gère un ensemble E de sommets dont les longueurs finales de plus court chemin à partir de l'origine s ont déjà été calculées. A chaque itération, l'algorithme choisit le sommet $u \in S - E$ dont l'estimation de plus court chemin est minimale, ajoute u à E , puis relâche tous les arcs partant de u . Dans l'implémentation ci-après, on gère une file de priorités min F de sommets, en prenant pour clé la valeur de leur attribut d .

```

DIJKSTRA( $G, w, s$ )
1 SOURCE-UNIQUE-INITIALISATION( $G, s$ )
2  $E \leftarrow \emptyset$ 
3  $F \leftarrow S[G]$ 
4 tant que  $F \neq \emptyset$ 
5   faire  $u \leftarrow \text{EXTRAIRE-MIN}(F)$ 
6      $E \leftarrow E \cup \{u\}$ 
7     pour chaque sommet  $v \in \text{Adj}[u]$ 
8       faire RELÂCHER( $u, v, w$ )

```

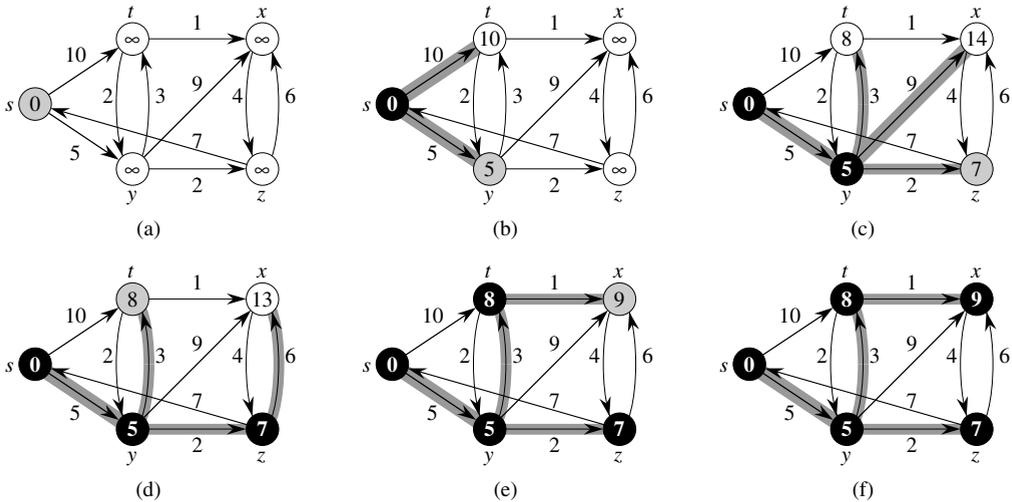


Figure 24.6 L'exécution de l'algorithme de Dijkstra. L'origine s est le sommet le plus à gauche. Les estimations de plus court chemin sont représentées à l'intérieur des sommets, et les arcs en gris indiquent les prédécesseurs. Les sommets noirs sont dans l'ensemble E , et les sommets blancs sont dans la file de priorités min $F = S - E$. (a) La situation juste avant la première itération de la boucle **tant que** des lignes 4–8. Le sommet gris contient la valeur minimale de d et est choisi comme sommet u à la ligne 5. (b)–(f) La situation après les itérations successives de la boucle **tant que**. A chaque étape, le sommet gris est celui choisi comme sommet u à la ligne 5 de l'itération suivante. Les valeurs d et π représentées en (f) sont les valeurs finales.

L'algorithme de Dijkstra relâche les arcs comme illustré à la figure 24.6. La ligne 1 effectue l'initialisation habituelle des valeurs d et π , et la ligne 2 initialise l'ensemble E . L'algorithme conserve l'invariant suivant : $F = S - E$ au début de chaque itération

de la boucle **tant que** des lignes 4-8. La ligne 3 initialise la file de priorités min F en y incluant tous les sommets de S ; comme à ce moment-là $E = \emptyset$, l'invariant est vrai après la ligne 3. A chaque passage dans la boucle **tant que** des lignes 4-8, un sommet u est extrait de $F = S - E$ et inséré dans l'ensemble E , ce qui a pour effet de conserver l'invariant. (Lors de la première itération, $u = s$.) Le sommet u possède donc la plus petite estimation de plus court chemin parmi tous les sommets de $S - E$. Ensuite, les lignes 7-8 relâchent chaque arc (u, v) partant de u , ce qui a pour effet de mettre à jour l'estimation $d[v]$ et le prédécesseur $\pi[v]$ si le plus court chemin jusqu'à v peut être amélioré en passant par u . Observez que les sommets ne sont jamais insérés dans F après la ligne 3 et que chaque sommet est extrait de F et inséré dans E exactement une fois, de sorte que la boucle **tant que** des lignes 4-8 est répétée exactement $|S|$ fois.

L'algorithme de Dijkstra choisissant toujours le sommet de $S - E$ « le moins coûteux », ou « le plus proche », pour l'insérer dans E , on dit qu'il utilise une stratégie gloutonne. Les stratégies gloutonnes sont présentées en détail au chapitre 16, mais sa lecture n'est pas indispensable pour comprendre le fonctionnement de l'algorithme de Dijkstra. Les stratégies gloutonnes ne conduisent pas toujours à des résultats optimaux de manière générale mais, comme le montrent le théorème suivant ainsi que son corollaire, l'algorithme de Dijkstra calcule effectivement les plus courts chemins. Le principe est de montrer que, chaque fois qu'un sommet u est inséré dans l'ensemble E , on a $d[u] = \delta(s, u)$.

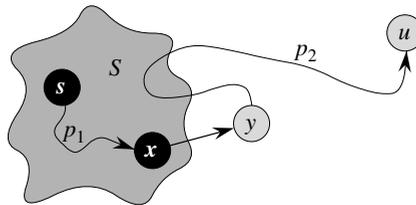


Figure 24.7 La démonstration du théorème 24.6. L'ensemble E est non vide juste avant qu'on y insère le sommet u . Un plus court chemin p de l'origine s vers le sommet u peut se décomposer en $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$, où y est le premier sommet du chemin à ne pas être dans E et $x \in E$ est le sommet qui précède immédiatement y . Les sommets x et y sont distincts, mais on pourrait avoir $s = x$ ou $y = u$. Le chemin p_2 peut ou non re-entrer dans l'ensemble E .

Théorème 24.6 (Validité de l'algorithme de Dijkstra) *Si l'on exécute l'algorithme de Dijkstra sur un graphe orienté pondéré $G = (S, A)$, avec une fonction de pondération positive w et une origine s , après exécution l'on a $d[u] = \delta(s, u)$ pour tous les sommets $u \in S$.*

Démonstration :

Nous utiliserons l'invariant de boucle que voici : Au début de chaque itération de la boucle **tant que** des lignes 4-8, $d[v] = \delta(s, v)$ pour chaque sommet $v \in E$.

Il suffit de montrer que, pour chaque sommet $u \in S$, on a $d[u] = \delta(s, u)$ au moment où u est ajouté à E . Lorsque nous aurons montré que $d[u] = \delta(s, u)$, nous nous appuierons sur la propriété propriété du majorant pour montrer qu'ensuite l'égalité reste toujours vraie.

Initialisation : Initialement, $E = \emptyset$ et donc l'invariant est trivialement vrai.

Conservation : On veut montrer que, à chaque itération, $d[u] = \delta(s, u)$ pour le sommet ajouté à l'ensemble E . Raisonnons par l'absurde : soit u le premier sommet pour lequel $d[u] \neq \delta(s, u)$ quand on l'ajoute à E . Concentrons-nous sur la situation au début de l'itération de la boucle **tant que** dans laquelle u est ajouté à E pour aboutir à la contradiction que $d[u] = \delta(s, u)$ à cet instant, en examinant un plus court chemin de s à u . On doit avoir $u \neq s$ car s est le premier sommet ajouté à E et $d[s] = \delta(s, s) = 0$ à cet instant. Comme $u \neq s$, on a aussi $E \neq \emptyset$ juste avant que u soit ajouté à E . Il doit y avoir un chemin de s à u , car sinon $d[u] = \delta(s, u) = \infty$ d'après la propriété aucun-chemin, ce qui contredirait notre hypothèse que $d[u] \neq \delta(s, u)$. Comme il y a au moins un chemin, il existe un plus court chemin p entre s et u . Avant que l'on ajoute u à E , le chemin p relie un sommet de E , à savoir s , à un sommet de $S - E$, à savoir u . Soit y le premier sommet de p tel que $y \in S - E$, et soit $x \in E$ le prédécesseur de y . Donc, comme illustré sur la figure 24.7, le chemin p peut se décomposer en $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$. (L'un ou l'autre des chemins p_1 ou p_2 peut ne pas avoir d'arcs.)

Nous affirmons que $d[y] = \delta(s, y)$ quand u est ajouté à E . Pour prouver cette affirmation, observons que $x \in E$. Alors, comme u est le premier sommet pour lequel $d[u] \neq \delta(s, u)$ quand on l'ajoute à E , on avait $d[x] = \delta(s, x)$ quand x a été ajouté à E . L'arc (x, y) était relâché à cet instant, et donc notre affirmation découle de la propriété de convergence.

On peut maintenant aboutir à une contradiction pour prouver que $d[u] = \delta(s, u)$. Comme y vient avant u sur un plus court chemin de s à u et que tous les poids d'arc sont positifs (entre autres, ceux du chemin p_2), on a $\delta(s, y) \leq \delta(s, u)$, et donc

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \quad (\text{d'après la propriété propriété du majorant}) . \end{aligned} \tag{24.2}$$

Mais comme les sommets u et y étaient dans $S - E$ quand u a été choisi en ligne 5, on a $d[u] \leq d[y]$. Ainsi, les deux inégalités de (24.2) sont en fait des égalités, ce qui donne

$$d[y] = \delta(s, y) = \delta(s, u) = d[u] .$$

Par conséquent, $d[u] = \delta(s, u)$, ce qui contredit notre choix de u . On en déduit que $d[u] = \delta(s, u)$ quand u est ajouté à E et que cette égalité reste ensuite vraie en permanence.

Terminaison : À la fin de l'exécution $F = \emptyset$, ce qui, combiné avec notre précédent invariant selon lequel $F = S - E$, implique que $E = S$. Donc, $d[u] = \delta(s, u)$ pour tous les sommets $u \in S$. □

Corollaire 24.7 *Si l'on exécute l'algorithme de Dijkstra sur un graphe orienté pondéré $G = (S, A)$ ayant la fonction de pondération positive w et l'origine s , à la fin de l'exécution le sous-graphe prédécesseur G_π est une arborescence de plus courts chemins de racine s .*

Démonstration : Immédiate d'après le théorème 24.6 et la propriété de sous-graphe prédécesseur. \square

a) Analyse

Quelle est la rapidité de l'algorithme de Dijkstra ? Il gère la file de priorités min F en appelant trois opérations de file de priorités : INSÉRER (implicite en ligne 3), EXTRAIRE-MIN (ligne 5) et DIMINUER-CLÉ (implicite dans RELÂCHER, qui est appelée en ligne 8). INSÉRER est appelée une fois par sommet, de même que EXTRAIRE-MIN. Comme chaque sommet $v \in S$ est inséré dans E une fois exactement, chaque arc de la liste d'adjacences $Adj[v]$ est examiné exactement une fois dans la boucle **pour** des lignes 7–8 pendant le déroulement de l'algorithme. Le nombre d'arcs total dans toutes les listes d'adjacence valant $|A|$, il existe au total $|A|$ itérations de cette boucle **pour** et donc il y a en tout au plus $|A|$ opérations DIMINUER-CLÉ. (Notez d'erechef que nous faisons ici aussi une analyse par agrégat.)

Le temps d'exécution de l'algorithme de Dijkstra dépend de la façon dont est mise en œuvre la file de priorité min. Considérons d'abord le cas où l'on gère la file en exploitant le fait que les sommets sont numérotés de 1 à $|S|$. On se contente de stocker $d[v]$ dans le v ème élément d'un tableau. Chaque opération INSÉRER et DIMINUER-CLÉ prend un temps $O(1)$, et chaque opération EXTRAIRE-MIN prend un temps $O(S)$ (car il faut parcourir tout le tableau), pour un temps total de $O(S^2 + A) = O(S^2)$.

Si le graphe est suffisamment peu dense, en particulier si $A = o(S^2 / \lg S)$, il est commode de mettre en œuvre la file de priorité min à l'aide d'un tas min binaire. (Comme vu à la section 6.5, un détail d'implémentation important est que les sommets et les éléments du tas correspondants doivent gérer des balises pointant vers les objets associés.) Chaque opération EXTRAIRE-MIN prend alors un temps $O(\lg S)$. Comme précédemment, il y a $|S|$ de telles opérations. Le temps de construction du tas min binaire est $O(S)$. Chaque opération DIMINUER-CLÉ prend un temps $O(\lg S)$, et ici aussi il y a au plus $|A|$ de telles opérations. Le temps d'exécution total est donc $O((S + A) \lg S)$, soit $O(A \lg S)$ si tous les sommets sont accessibles depuis l'origine. Ce temps d'exécution est une amélioration par rapport à l'implémentation directe à temps $O(S^2)$ si $A = o(S^2 / \lg S)$.

On peut en fait atteindre un temps d'exécution $O(S \lg S + A)$ en implémentant la file de priorité min à l'aide d'un tas de Fibonacci (voir chapitre 20). Le coût amorti de chacune des $|S|$ opérations EXTRAIRE-MIN est $O(\lg S)$, et chacun des $|A|$ (au plus) appels DIMINUER-CLÉ ne consomme que $O(1)$ de temps amorti. Historiquement, le développement des tas de Fibonacci fut motivé par l'observation suivante : dans l'algorithme de Dijkstra, il y a généralement beaucoup plus d'appels DIMINUER-CLÉ que d'appels EXTRAIRE-MIN, de sorte qu'une méthode permettant de ramener le temps amorti de chaque opération DIMINUER-CLÉ à $o(\lg S)$, sans augmenter le temps amorti de EXTRAIRE-MIN, donnera une implémentation plus rapide asymptotiquement que les tas binaires.

L'algorithme de Dijkstra révèle quelques similitudes tant avec le parcours en largeur (voir section 22.2) qu'avec l'algorithme de Prim calculant les arbres couvrants de poids minimum (voir section 23.2). Il ressemble au parcours en largeur au sens où l'ensemble E correspond à l'ensemble des sommets noirs d'un parcours en largeur ; de même qu'un sommet de E a son poids final égale à la longueur du plus court chemin, de même chaque sommet noir du parcours en largeur a sa distance correcte en largeur. L'algorithme de Dijkstra ressemble à celui de Prim au sens où tous les deux font appel à une file de priorité min pour trouver le sommet « le moins coûteux » situé en dehors d'un ensemble donné (l'ensemble E pour l'algorithme de Dijkstra et l'arbre en cours de construction pour l'algorithme de Prim), insèrent ce sommet dans l'ensemble, puis ajustent en conséquence les poids des autres sommets situés en dehors de l'ensemble.

Exercices

24.3.1 Exécuter l'algorithme de Dijkstra sur le graphe orienté de la figure 24.2, en prenant d'abord comme origine le sommet s , puis le sommet z . En s'inspirant de la figure 24.6, donner la valeur des attributs d et π ainsi que les sommets de l'ensemble E après chaque itération de la boucle **tant que**.

24.3.2 Donner un exemple simple de graphe orienté comportant des arcs de poids négatif, pour lequel l'algorithme de Dijkstra donne des réponses incorrectes. Pourquoi la démonstration du théorème 24.6 n'est-elle plus valable quand on autorise des arcs de poids négatif ?

24.3.3 Supposons que la ligne 4 de l'algorithme de Dijkstra soit remplacée par celle-ci :

4 **tant que** $|F| > 1$

Cette modification permet à la boucle **tant que** de s'exécuter $|S| - 1$ fois au lieu de $|S|$ fois. L'algorithme ainsi modifié est-il correct ?

24.3.4 Soit un graphe orienté $G = (S, A)$ pour lequel chaque arc $(u, v) \in A$ possède une valeur $r(u, v)$ réelle vérifiant $0 \leq r(u, v) \leq 1$. Cette valeur représente la fiabilité du canal de communication entre le sommet u et le sommet v . On interprète $r(u, v)$ comme étant la probabilité pour que le canal entre u et v ne soit pas interrompu, et on suppose que ces probabilités sont indépendantes. Donner un algorithme efficace permettant de trouver le chemin le plus fiable entre deux sommets donnés.

24.3.5 Soit $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \{0, 1, \dots, W - 1\}$ pour un certain entier positif W , tel que deux sommets n'aient jamais la même longueur de plus court chemin à partir de l'origine s . On suppose maintenant que l'on définit un graphe orienté non pondéré $G' = (S \cup S', A')$, en remplaçant chaque arc $(u, v) \in A$ par une suite de $w(u, v)$ arcs de poids unitaire. Combien de sommets possède G' ? On suppose maintenant que l'on exécute une recherche en largeur sur G' . Montrer que l'ordre dans lequel les sommets de S sont coloriés en noir dans le parcours en largeur de G' est le même que l'ordre dans lequel les sommets de S sont extraits de la file de priorité en ligne 5 de DIJKSTRA quand on l'exécute sur G .

24.3.6 Soit $G = (S, A)$ un graphe orienté pondéré de fonction de pondération $w : E \rightarrow \{0, 1, \dots, W\}$, pour un certain entier positif W . Modifier l'algorithme de Dijkstra pour qu'il calcule les plus courts chemins issus d'une origine donnée s en temps $O(W|S| + |A|)$.

24.3.7 Modifier l'algorithme trouvé à l'exercice 24.3.6 pour qu'il tourne en $O(|S| + |A| \lg W)$. (*Conseil* : Combien d'estimations de plus court chemin distinctes peut-il y avoir dans $S - E$ à n'importe quel moment ?)

24.3.8 Soit un graphe orienté pondéré $G = (S, A)$ dans lequel les arcs qui partent du sommet origine s peuvent avoir des poids négatifs, tous les autres poids d'arc sont positifs et il n'y a pas de circuit de longueur strictement négative. Montrer que l'algorithme de Dijkstra trouve bien les plus courts chemins issus de s .

24.4 CONTRAINTES DE POTENTIEL ET PLUS COURTS CHEMINS

Dans le problème général de la programmation linéaire qui sera étudié au chapitre 29, on souhaite optimiser une fonction linéaire soumise à un ensemble d'inégalités linéaires. Dans cette section, on s'intéressera à un cas particulier de programmation linéaire, qui peut être ramené à trouver les plus courts chemins à partir d'une origine unique. Le problème des plus courts chemins à origine unique qui en résulte peut alors être résolu grâce à l'algorithme de Bellman-Ford, ce qui résout par la même occasion le problème de programmation linéaire.

a) Programmation linéaire

Dans le **problème de la programmation linéaire** général, on dispose en entrée d'une matrice A $m \times n$, d'un vecteur b à m composantes et d'un vecteur c à n composantes. On souhaite trouver un vecteur x à n éléments qui maximise la **fonction objectif** $\sum_{i=1}^n c_i x_i$, où x vérifie les m contraintes données par $Ax \leq b$.

L'algorithme du simplexe, qui constitue l'essentiel du chapitre 29, ne s'exécute pas toujours en temps polynomial (exprimé en fonction de la taille de l'entrée), mais d'autres algorithmes de programmation linéaire s'exécutent en temps polynomial. Pour plusieurs raisons, il est important de comprendre l'organisation d'un problème de programmation linéaire. D'abord, si l'on sait qu'un problème donné peut se ramener à un problème de programmation linéaire de taille polynomiale, on peut immédiatement en déduire qu'il existe un algorithme en temps polynomial pour résoudre le problème de départ. Ensuite, il existe de nombreux cas particuliers de programmation linéaire, pour lesquels on peut trouver des algorithmes plus rapides. Par exemple, comme le montre cette section, le problème du plus court chemin à origine unique est un cas particulier de programmation linéaire. D'autres problèmes peuvent être ramenés à de la programmation linéaire, notamment le problème du plus court chemin

pour un couple de sommets donné (exercice 24.4.4) et le problème du flot maximal (exercice 26.1.8).

Il arrive que l'on ne s'intéresse pas vraiment à la fonction objectif; on souhaite tout simplement trouver une **solution réalisable**, c'est-à-dire un vecteur x qui vérifie $Ax \leq b$, ou bien déterminer qu'aucune solution réalisable n'existe. Nous allons nous intéresser particulièrement à l'un de ces **problèmes d'existence**.

b) Systèmes de contraintes de potentiel

Dans un **système de contraintes de potentiel**, chaque ligne de la matrice A du programme linéaire contient un 1 et un -1 , et toutes les autres composantes de A valent 0. Les contraintes données par $Ax \leq b$ représentent donc un ensemble de m **contraintes de potentiel** à n inconnues, de la forme

$$x_j - x_i \leq b_k,$$

où $1 \leq i, j \leq n$ et $1 \leq k \leq m$.

Considérons par exemple le problème de la détermination d'un vecteur $x = (x_i)$ à cinq composantes, sous les contraintes

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}.$$

Ce problème équivaut à trouver les inconnues x_i , pour $i = 1, 2, \dots, 5$, telles que les 8 contraintes de potentiel suivantes soient vérifiées :

$$x_1 - x_2 \leq 0, \quad (24.3)$$

$$x_1 - x_5 \leq -1, \quad (24.4)$$

$$x_2 - x_5 \leq 1, \quad (24.5)$$

$$x_3 - x_1 \leq 5, \quad (24.6)$$

$$x_4 - x_1 \leq 4, \quad (24.7)$$

$$x_4 - x_3 \leq -1, \quad (24.8)$$

$$x_5 - x_3 \leq -3, \quad (24.9)$$

$$x_5 - x_4 \leq -3. \quad (24.10)$$

Une solution à ce problème est $x = (-5, -3, 0, -1, -4)$, comme on peut le vérifier directement en testant chaque inégalité. En fait, il existe plusieurs solutions à ce problème. $x' = (0, 2, 5, 4, 1)$ en est une autre. Ces deux solutions sont liées : chaque

composante de x' vaut 5 de plus que la composante correspondante de x . Ce fait n'est pas une simple coïncidence.

Lemme 24.8 Soit $x = (x_1, x_2, \dots, x_n)$ une solution au système $Ax \leq b$ de contraintes de potentiel, et soit d une constante quelconque. Alors $x + d = (x_1 + d, x_2 + d, \dots, x_n + d)$ est également solution de $Ax \leq b$.

Démonstration : Pour chaque x_i et x_j , on a $(x_j + d) - (x_i + d) = x_j - x_i$. Donc, si x vérifie $Ax \leq b$, $x + d$ aussi. \square

Les systèmes de contraintes de potentiel admettent de nombreuses applications différentes. Par exemple, les inconnues x_i peuvent représenter des dates auxquelles ont lieu des événements. Chaque contrainte peut être vue comme une règle établissant qu'un événement ne peut pas apparaître trop tard après un autre événement. Les événements pourraient par exemple représenter les travaux à effectuer pour la construction d'une maison. Si le creusement des fondations commence à l'instant x_1 et prend 3 jours, et si la coulée du béton pour les fondations commence à l'instant x_2 , on souhaite que $x_2 \geq x_1 + 3$ ou, ce qui est équivalent, $x_1 - x_2 \leq -3$.

c) Graphes de potentiel

Il est utile d'interpréter les systèmes de contraintes de potentiel du point de vue de la théorie des graphes. Le principe est que pour un système $Ax \leq b$ de contraintes de potentiel, la matrice A $m \times n$ du programme linéaire peut être interprétée comme une matrice d'incidences (voir exercice 22.1.7) d'un graphe à n sommets et m arcs. Chaque sommet v_i du graphe, pour $i = 1, 2, \dots, n$, correspond à une des n inconnues x_i . Chaque arc du graphe correspond à l'une des m inégalités à deux inconnues.

Plus formellement, étant donné un système $Ax \leq b$ de contraintes de potentiel, le **graphe de potentiel** est un graphe orienté pondéré $G = (S, A)$, où

$$S = \{v_0, v_1, \dots, v_n\}$$

et

$$A = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ est une contrainte}\} \\ \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \dots, (v_0, v_n)\} .$$

Le sommet supplémentaire v_0 est ajouté, comme nous le verrons bientôt, pour garantir que chaque autre sommet est accessible à partir de v_0 . Donc, l'ensemble S des sommets est constitué de l'union des sommets v_i associés aux variables x_i et du sommet v_0 . L'ensemble A des arcs contient un arc pour chaque contrainte de potentiel, plus un arc (v_0, v_i) pour chaque inconnue x_i . Si $x_j - x_i \leq b_k$ est une contrainte de potentiel, alors le poids de l'arc (v_i, v_j) sera $w(v_i, v_j) = b_k$. Le poids de chaque arc sortant de v_0 vaut 0. La figure 24.8 montre le graphe de potentiel correspondant au système (24.3)–(24.10) de contraintes de potentiel.

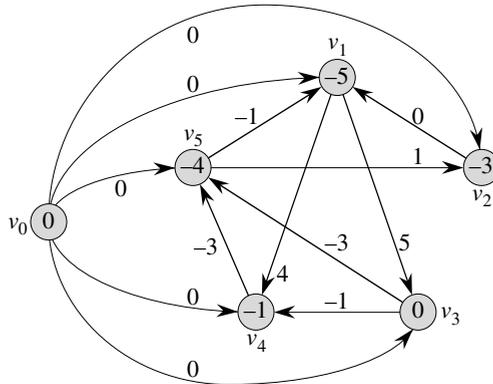


Figure 24.8 Le graphe de potentiel correspondant au système (24.3)–(24.10) de contraintes de potentiel. La valeur de $\delta(v_0, v_i)$ est montrée pour chaque sommet v_i . Une solution réalisable pour le système est $x = (-5, -3, 0, -1, -4)$.

Le théorème suivant montre qu’une solution d’un système de contraintes de potentiel peut être obtenue en recherchant les longueurs de plus court chemin dans le graphe de potentiel correspondant.

Théorème 24.9 *Étant donné un système $Ax \leq b$ de contraintes de potentiel, soit $G = (S, A)$ le graphe de potentiel correspondant. Si G ne contient aucun circuit de longueur strictement négative, alors*

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \dots, \delta(v_0, v_n)) \tag{24.11}$$

est une solution réalisable pour le système. Si G contient un circuit de longueur strictement négative, il n’existe aucune solution réalisable pour le système.

Démonstration : Commençons par montrer que si le graphe de potentiel ne contient aucun circuit de longueur strictement négative, l’équation (24.11) donne une solution réalisable. Considérons un arc $(v_i, v_j) \in A$ quelconque. D’après l’inégalité triangulaire, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$, ou encore $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Donc, si l’on pose $x_i = \delta(v_0, v_i)$ et $x_j = \delta(v_0, v_j)$, la contrainte de potentiel $x_j - x_i \leq w(v_i, v_j)$ correspondant à l’arc (v_i, v_j) est vérifiée.

Montrons à présent que si le graphe de potentiel contient un circuit de longueur strictement négative, le système de contraintes de potentiel ne possède aucune solution réalisable. Sans perte de généralité, appelons $c = \langle v_1, v_2, \dots, v_k \rangle$ un circuit de longueur strictement négative, où $v_1 = v_k$. (Le sommet v_0 ne peut pas se trouver sur le circuit c , car il ne possède aucun arc entrant.) Le circuit c correspond aux contraintes de potentiel suivantes :

$$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2), \\ x_3 - x_2 &\leq w(v_2, v_3), \\ &\vdots \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k), \\ x_1 - x_k &\leq w(v_k, v_1). \end{aligned}$$

Une solution pour x qui vérifierait chacune de ces k inégalités devrait aussi vérifier l'inégalité obtenue en les sommant. Si l'on somme les membres de gauche, chaque inconnue x_i est ajoutée une fois et soustraite une fois, de sorte que le membre gauche de la somme est égal à 0. Le membre droit de la somme est égal à $w(c)$, et l'on obtient donc $0 \leq w(c)$. Mais comme c est un circuit de longueur strictement négative, $w(c) < 0$, ce qui signifie que toutes les solutions pour x doivent vérifier $0 \leq w(c) < 0$, ce qui est impossible. \square

d) Résolution de systèmes de contraintes de potentiel

Le théorème 24.9 dit qu'il est possible d'utiliser l'algorithme de Bellman-Ford pour résoudre un système de contraintes de potentiel. Puisqu'il existe des arcs sortant du sommet origine v_0 vers tous les autres sommets du graphe de potentiel, tout circuit de longueur strictement négative dans le graphe de potentiel est accessible à partir de v_0 . Si l'algorithme de Bellman-Ford retourne VRAI, alors les longueurs des plus courts chemins donnent la solution réalisable pour le système. À la figure 24.8, par exemple, les longueurs des plus courts chemins donnent la solution réalisable $x = (-5, -3, 0, -1, -4)$, et d'après le lemme 24.8, $x = (d - 5, d - 3, d, d - 1, d - 4)$ est également une solution réalisable pour une constante quelconque d . Si l'algorithme de Bellman-Ford retourne FAUX, il n'existe aucune solution réalisable pour le système de contraintes de potentiel.

Un système de contraintes de potentiel à m contraintes et n inconnues correspond à un graphe à $n + 1$ sommets et $n + m$ arcs. Ainsi, à l'aide de l'algorithme de Bellman-Ford, il est possible de résoudre le système en temps $O((n + 1)(n + m)) = O(n^2 + nm)$. L'exercice 24.4.5 vous demandera de montrer que l'algorithme s'exécute en réalité en temps $O(nm)$, même si m est très inférieur à n .

Exercices

24.4.1 Trouver une solution réalisable ou montrer qu'il n'existe aucune solution pour le système de contraintes de potentiel suivant :

$$\begin{aligned} x_1 - x_2 &\leq 1, \\ x_1 - x_4 &\leq -4, \\ x_2 - x_3 &\leq 2, \\ x_2 - x_5 &\leq 7, \\ x_2 - x_6 &\leq 5, \\ x_3 - x_6 &\leq 10, \\ x_4 - x_2 &\leq 2, \\ x_5 - x_1 &\leq -1, \\ x_5 - x_4 &\leq 3, \\ x_6 - x_3 &\leq -8. \end{aligned}$$

24.4.2 Trouver une solution réalisable ou montrer qu'il n'existe aucune solution pour le système de contraintes de potentiel suivant :

$$\begin{aligned}
 x_1 - x_2 &\leq 4, \\
 x_1 - x_5 &\leq 5, \\
 x_2 - x_4 &\leq -6, \\
 x_3 - x_2 &\leq 1, \\
 x_4 - x_1 &\leq 3, \\
 x_4 - x_3 &\leq 5, \\
 x_4 - x_5 &\leq 10, \\
 x_5 - x_3 &\leq -4, \\
 x_5 - x_4 &\leq -8.
 \end{aligned}$$

24.4.3 La longueur d'un plus court chemin partant du sommet supplémentaire v_0 dans un graphe de potentiel peut-il être positif ? Dire pourquoi.

24.4.4 Exprimer le problème du plus court chemin à paire unique en tant que programme linéaire.

24.4.5 Montrer comment modifier légèrement l'algorithme de Bellman-Ford pour que, quand on l'utilise pour résoudre un système de contraintes de potentiel à m inégalités sur n inconnues, le temps d'exécution soit $O(nm)$.

24.4.6 Supposez que, en plus d'un système de contraintes de différences (potentiels), on veuille gérer des *contraintes d'égalité* de la forme $x_i = x_j + b_k$. Montrer comment adapter l'algorithme de Bellman-Ford pour qu'il résolve ce type de système de potentiel.

24.4.7 Montrer comment un système de contraintes de potentiel peut être résolu par un algorithme style Bellman-Ford qui s'exécute sur un graphe de potentiel sans le sommet supplémentaire v_0 .

24.4.8 * Soit $Ax \leq b$ un système de m contraintes de potentiel à n inconnues. Montrer que l'algorithme de Bellman-Ford, quand il est exécuté sur le graphe de potentiel correspondant, maximise la fonction $\sum_{i=1}^n x_i$ sous les contraintes $Ax \leq b$ et $x_i \leq 0$ pour tout x_i .

24.4.9 * Montrer que l'algorithme de Bellman-Ford, quand il est exécuté sur le graphe de potentiel correspondant à un système $Ax \leq b$ de contraintes de potentiel, minimise la quantité $(\max \{x_i\} - \min \{x_i\})$, sous la contrainte $Ax \leq b$. Expliquer en quoi ce fait pourrait se révéler intéressant si l'algorithme servait à planifier des travaux de construction.

24.4.10 On suppose que chaque ligne de la matrice A d'un programme linéaire $Ax \leq b$ correspond soit à une contrainte de potentiel, soit à une contrainte à variable unique de la forme $x_i \leq b_k$, soit à une contrainte à variable unique de la forme $-x_i \leq b_k$. Montrer que l'algorithme de Bellman-Ford peut être adapté pour résoudre ce type de système de contraintes.

24.4.11 Donner un algorithme efficace permettant de résoudre un système $Ax \leq b$ de contraintes de potentiel lorsque tous les éléments de b sont à valeurs réelles et toutes les inconnues x_i sont à valeurs entières.

24.4.12 * Donner un algorithme efficace permettant de résoudre un système $Ax \leq b$ de contraintes de potentiel lorsque tous les éléments de b sont à valeurs réelles et qu'un sous-ensemble spécifié de certaines des inconnues x_i (mais pas nécessairement de toutes) sont à valeurs entières.

24.5 DÉMONSTRATIONS DES PROPRIÉTÉS DE PLUS COURT CHEMIN

Dans tout ce chapitre, nous avons démontré la conformité de nos procédures en nous appuyant sur les propriétés d'inégalité triangulaire, propriété du majorant, propriété aucun-chemin, propriété de convergence, propriété de relâchement de chemin et propriété de sous-graphe prédécesseur, énoncées sans démonstration en début de chapitre.. Dans cette section, nous allons prouver tout cela.

a) Propriété inégalité triangulaire

Lors de l'étude du parcours en largeur (section 22.2), nous avons démontré sous la forme du lemme 22.1 une propriété simple des plus courtes distances dans les graphes non pondérés. La propriété d'inégalité triangulaire généralise ce résultat aux graphes pondérés.

Lemme 24.10 (Inégalité triangulaire) *Soit $G = (S, A)$ un graphe orienté pondéré, ayant pour fonction de pondération $w : A \rightarrow \mathbf{R}$ et pour sommet origine s . Alors, pour tout arc $(u, v) \in A$, on a*

$$\delta(s, v) \leq \delta(s, u) + w(u, v) .$$

Démonstration : Supposons qu'il y ait un plus court chemin p entre l'origine s et le sommet v . Alors, p n'est pas plus long que n'importe quel autre chemin entre s et v . Plus spécialement, le chemin p n'est pas plus long que le chemin particulier qui emprunte un plus court chemin de s au sommet u puis qui emprunte l'arc (u, v) .

L'exercice 24.5.3 vous demandera de traiter le cas où il n'y a pas de plus court chemin entre s et v . □

b) Effets du relâchement sur les estimations de plus court chemin

Les lemmes suivants montrent comment sont modifiées les estimations de plus court chemin quand on exécute une suite d'étapes de relâchement sur les arcs d'un graphe orienté pondéré qui a été initialisé par SOURCE-UNIQUE-INITIALISATION.

Lemme 24.11 (Propriété du majorant) *Soit $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$. Soit $s \in S$ le sommet origine ; on suppose que le graphe est initialisé par SOURCE-UNIQUE-INITIALISATION(G, s). Alors, $d[v] \geq \delta(s, v)$ pour tout $v \in S$, et cet invariant est conservé pour toute séquence d'étapes de relâchement sur les arcs de G . De plus, une fois que $d[v]$ a atteint son minorant $\delta(s, v)$, il n'est plus jamais modifié.*

Démonstration : Nous allons prouver l'invariant $d[v] \geq \delta(s, v)$ pour tout $v \in S$ en raisonnant par récurrence sur le nombre d'étapes de relâchement. L'invariant $d[v] \geq \delta(s, v)$ est vérifié après l'initialisation, puisque $d[s] = 0 \geq \delta(s, s)$ (notez que $\delta(s, s)$ vaut $-\infty$ si s se trouve sur un circuit de longueur strictement négative, et 0 sinon) et $d[v] = \infty$ implique $d[v] \geq \delta(s, v)$ pour tout $v \in S - \{s\}$.

Considérons donc le relâchement d'un arc (u, v) . L'hypothèse de récurrence entraîne que $d[x] \geq \delta(s, x)$ pour tout $x \in S$ avant le relâchement. La seule valeur d susceptible de changer est $d[v]$. Si elle change, on a

$$\begin{aligned} d[v] &= d[u] + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{(d'après l'hypothèse de récurrence)} \\ &\geq \delta(s, v) && \text{(d'après la propriété d'inégalité triangulaire)} . \end{aligned}$$

et donc l'invariant est conservé.

Pour comprendre pourquoi la valeur de $d[v]$ n'est plus modifiée une fois que $d[v] = \delta(s, v)$, il suffit de remarquer qu'après avoir atteint sa borne inférieure, $d[v]$ ne peut plus diminuer puisque nous venons de montrer que $d[v] \geq \delta(s, v)$, et qu'il ne peut plus augmenter puisque les étapes de relâchement n'augmentent pas les valeurs d . \square

Corollaire 24.12 (Propriété aucun-chemin) *On suppose que, dans un graphe orienté pondéré $G = (S, A)$ de fonction de pondération $w : A \rightarrow \mathbf{R}$, aucun chemin ne relie un sommet origine $s \in S$ à un sommet donné $v \in S$. Alors, après initialisation du graphe par SOURCE-UNIQUE-INITIALISATION(G, s), on a $d[v] = \delta(s, v)$, et cette inégalité est maintenue comme invariant par n'importe quelle séquence d'étapes de relâchement sur les arcs de G .*

Démonstration : D'après la propriété de majorant (lemme 24.11), on a toujours $\infty = \delta(s, v) \leq d[v]$, d'où $d[v] = \infty = \delta(s, v)$. \square

Lemme 24.13 *Soit $G = (S, A)$ un graphe orienté pondéré ayant pour fonction de pondération $w : A \rightarrow \mathbf{R}$, et soit $(u, v) \in A$. Alors, juste après relâchement de l'arc (u, v) via exécution de RELÂCHER(u, v, w), on a $d[v] \leq d[u] + w(u, v)$.*

Démonstration : Si, juste avant le relâchement de l'arc (u, v) , on a $d[v] > d[u] + w(u, v)$, alors $d[v] = d[u] + w(u, v)$ après. Si, en revanche, $d[v] \leq d[u] + w(u, v)$ juste avant le relâchement, alors ni $d[u]$ ni $d[v]$ ne change, et donc $d[v] \leq d[u] + w(u, v)$ après. \square

Lemme 24.14 (Propriété de convergence) *Soit $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$, soit $s \in S$ un sommet origine, et soit*

$s \rightsquigarrow u \rightarrow v$ un plus court chemin dans G pour deux sommets $u, v \in S$ donnés. On suppose que G est initialisé par SOURCE-UNIQUE-INITIALISATION(G, s) et qu'ensuite on exécute une séquence d'étapes de relâchement sur les arcs de G , qui inclut l'appel RELÂCHER(u, v, w). Si $d[u] = \delta(s, u)$ à un moment quelconque précédant l'appel, alors on aura toujours $d[v] = \delta(s, v)$ après l'appel.

Démonstration : D'après la propriété de majorant (lemme 24.11), si $d[u] = \delta(s, u)$ à un moment donné précédant le relâchement de l'arc (u, v) , cette inégalité est vérifiée par la suite. En particulier, après relâchement de l'arc (u, v) , on a

$$\begin{aligned} d[v] &\leq d[u] + w(u, v) && \text{(d'après le lemme 24.13)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(d'après le lemme 24.1).} \end{aligned}$$

D'après la propriété de majorant, $d[v] \geq \delta(s, v)$ d'où l'on conclut que $d[v] = \delta(s, v)$, et que cette égalité est maintenue ensuite. \square

Lemme 24.15 (Propriété de relâchement de chemin) *Soit $G = (S, A)$ un graphe orienté pondéré de fonction de pondération $w : A \rightarrow \mathbf{R}$, et soit $s \in S$ un sommet origine. Soit un plus court chemin $p = \langle v_0, v_1, \dots, v_k \rangle$ de $s = v_0$ vers v_k . Si G est initialisé par SOURCE-UNIQUE-INITIALISATION(G, s) puis que l'on applique une suite d'étapes de relâchement sur les arcs pris dans l'ordre $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, alors $d[v_k] = \delta(s, v_k)$ après ces relâchements et tout le temps ensuite. Cette propriété reste vraie même s'il y a d'autres relâchements d'arc, y compris des relâchements entremêlés avec les relâchements des arcs de p .*

Démonstration : Nous allons montrer par récurrence que, après que le i ème arc de p a été relâché, on a $d[v_i] = \delta(s, v_i)$. Pour la base, $i = 0$, et avant relâchement d'un quelconque arc de p , on a, d'après l'initialisation, $d[v_0] = d[s] = 0 = \delta(s, s)$. D'après la propriété, la valeur de $d[s]$ ne change plus après l'initialisation.

Pour l'étape inductive, supposons que $d[v_{i-1}] = \delta(s, v_{i-1})$, et examinons le relâchement de l'arc (v_{i-1}, v_i) . D'après la propriété, après ce relâchement, on a $d[v_i] = \delta(s, v_i)$, et cette égalité est conservée tout le temps ensuite. \square

c) Relâchement et arborescences de plus courts chemins

Nous allons montrer maintenant que, une fois qu'une suite de relâchements a forcé les estimations de plus court chemin à converger vers les longueurs de plus court chemin, le sous-graphe prédécesseur G_π induit par les valeurs π résultantes est une arborescence de plus courts chemins pour G . On commencera par le lemme suivant, qui montre que le sous-graphe prédécesseur forme une arborescence dont la racine est le sommet origine.

Lemme 24.16 *Soient $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$ et un sommet origine $s \in S$; supposons que G ne contienne*

aucun circuit de longueur strictement négative accessible depuis s . Alors, après initialisation du graphe par SOURCE-UNIQUE-INITIALISATION(G, s), le sous-graphe prédécesseur G_π forme une arborescence de racine s , et cette propriété est conservée en tant qu'invariant par toute séquence d'étapes de relâchement sur les arcs de G .

Démonstration : Initialement, le seul sommet de G_π est le sommet origine, et le lemme est trivial. Considérons un sous-graphe prédécesseur G_π obtenu après une séquence d'étapes de relâchement. Commençons par démontrer que G_π est sans circuit. Par l'absurde, supposons qu'une étape de relâchement crée un circuit dans le graphe G_π . Appelons ce circuit $c = \langle v_0, v_1, \dots, v_k \rangle$, où $v_k = v_0$. Alors, $\pi[v_i] = v_{i-1}$ pour $i = 1, 2, \dots, k$ et on peut supposer sans perte de généralité que l'apparition du circuit dans G_π était due au relâchement de l'arc (v_{k-1}, v_k) .

Nous affirmons que tous les sommets du circuit c sont accessibles depuis l'origine s . Pourquoi ? Chaque sommet de c a un prédécesseur différent de NIL ; chaque sommet a donc reçu, en même temps qu'une valeur π différente de NIL, une estimation de plus court chemin. D'après la propriété de majorant (lemme 24.11), chaque sommet du circuit c possède une longueur de plus court chemin finie, ce qui implique qu'il est accessible depuis s .

Nous allons examiner les estimations de plus court chemin sur c juste avant l'appel RELÂCHER(v_{k-1}, v_k, w) et montrer que c est un circuit de longueur strictement négative, ce qui contredit l'hypothèse selon laquelle G ne contient aucun circuit de longueur strictement négative accessible à partir de l'origine. Juste avant l'appel, on a $\pi[v_i] = v_{i-1}$ pour $i = 1, 2, \dots, k-1$. Donc, pour $i = 1, 2, \dots, k-1$, la dernière mise à jour de $d[v_i]$ était due à l'affectation $d[v_i] \leftarrow d[v_{i-1}] + w(v_i, v_{i-1})$. Si $d[v_{i-1}]$ a été modifié depuis, il a diminué. Par suite, juste avant l'appel RELÂCHER(v_{k-1}, v_k, w), on a

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i) \quad \text{pour tout } i = 1, 2, \dots, k-1. \quad (24.12)$$

Puisque $\pi[v_k]$ est modifié par l'appel, immédiatement avant, on a aussi l'inégalité stricte

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k).$$

La sommation de cette inégalité stricte avec les $k-1$ inégalités (24.12) donne la somme des estimations de plus court chemin le long du cycle c :

$$\begin{aligned} \sum_{i=1}^k d[v_i] &> \sum_{i=1}^k (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i). \end{aligned}$$

Mais

$$\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}],$$

puisque chaque sommet du circuit c apparaît exactement une fois dans chaque sommation. Cette égalité implique

$$0 > \sum_{i=1}^k w(v_{i-1}, v_i).$$

Donc, la somme des poids le long du circuit c est strictement négative, ce qui fournit la contradiction attendue.

Nous avons maintenant prouvé que G_π est un graphe orienté sans circuit. Pour montrer qu'il forme une arborescence de racine s , il suffit (voir exercice B.5.2) de prouver que, pour chaque sommet $v \in S_\pi$, il existe un chemin unique de s vers v dans G_π .

Il faut commencer par montrer qu'il existe un chemin partant de s vers chaque sommet de S_π . S_π est constitué des sommets dont le champ π est différent de NIL, plus le sommet s . Le principe est de démontrer par récurrence qu'il existe un chemin de s vers tous les sommets de S_π . Les détails sont laissés en exercice (voir exercice 24.5.6).

Pour compléter la démonstration du lemme, il faut maintenant montrer que, pour tout sommet $v \in S_\pi$, il existe au plus un chemin de s à v dans le graphe G_π . Prenons l'hypothèse inverse. Autrement dit, supposons qu'il existe deux chemins élémentaires reliant s à un certain sommet v : p_1 , qui peut se décomposer en $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, et p_2 , qui peut se décomposer en $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, où $x \neq y$. (Voir figure 24.9.) Mais alors, $\pi[z] = x$ et $\pi[z] = y$, ce qui implique la contradiction que $x = y$. On en conclut qu'il existe un chemin élémentaire unique dans G_π de s vers v , et donc que G_π forme une arborescence de racine s . \square

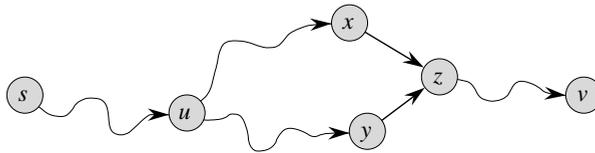


Figure 24.9 Illustration de l'unicité dans G_π d'un chemin reliant l'origine s au sommet v . S'il existait deux chemins p_1 ($s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$) et p_2 ($s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$), où $x \neq y$, alors $\pi[z] = x$ et $\pi[z] = y$, ce qui contredit l'hypothèse.

On peut montrer à présent que si, après une séquence d'étapes de relâchement, tous les sommets ont reçu leurs longueurs de plus court chemin réels, alors le sous-graphe prédécesseur G_π est une arborescence de plus courts chemins.

Lemme 24.17 (Propriété de sous-graphe prédécesseur) *Soient $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$ et un sommet origine $s \in S$; supposons que G ne contienne aucun circuit de longueur strictement négative accessible depuis s . Si on appelle SOURCE-UNIQUE-INITIALISATION(G, s), puis que l'on exécute une séquence quelconque d'étapes de relâchement sur les arcs de G qui produise $d[v] = \delta(s, v)$ pour tout $v \in S$, alors le sous-graphe prédécesseur G_π est une arborescence de plus courts chemins de racine s .*

Démonstration : Il faut démontrer que les trois propriétés d'arborescence de plus courts chemins données à la page 568 sont vérifiées par G_π . Pour la première propriété, on doit montrer que S_π est l'ensemble des sommets accessibles à partir de s . Par définition, un poids de plus court chemin $\delta(s, v)$ est fini si et seulement si v est accessible à partir de s , et donc les sommets accessibles depuis s sont exactement

ceux dont l'attribut d a une valeur finie. Mais $d[v]$ a une valeur finie pour un sommet $v \in S - \{s\}$ si et seulement si $\pi[v] \neq \text{NIL}$. Donc, les sommets de S_π sont exactement ceux accessibles à partir de s .

La deuxième propriété se déduit directement du lemme 24.16.

Il reste donc à démontrer la dernière propriété d'arborescence de plus courts chemins : pour tout $v \in S_\pi$, le chemin simple unique $s \xrightarrow{p} v$ dans G_π est un plus court chemin entre s et v dans G . Soit $p = \langle v_0, v_1, \dots, v_k \rangle$, où $v_0 = s$ et $v_k = v$. Pour $i = 1, 2, \dots, k$, on a à la fois $d[v_i] = \delta(s, v_i)$ et $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, d'où l'on conclut que $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. En sommant les poids le long du chemin p , on obtient

$$\begin{aligned} w(p) &= \sum_{i=1}^k w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^k (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) \quad (\text{pour cause de télescopage des termes}) \\ &= \delta(s, v_k) \quad (\text{car } \delta(s, v_0) = \delta(s, s) = 0) . \end{aligned}$$

Donc, $w(p) \leq \delta(s, v_k)$. Comme $\delta(s, v_k)$ est une borne inférieure de la longueur d'un chemin quelconque de s à v_k , on conclut que $w(p) = \delta(s, v_k)$ et donc que p est un plus court chemin de s vers $v = v_k$. \square

Exercices

24.5.1 Donner deux arborescences des plus courts chemins pour le graphe orienté de la figure 24.2, autres que les deux représentés.

24.5.2 Donner un exemple de graphe orienté pondéré $G = (S, A)$, de fonction de pondération $w : A \rightarrow \mathbf{R}$ et d'origine s , tel qui satisfasse à la propriété suivante : Pour tout arc $(u, v) \in A$, il existe une arborescence de plus courts chemins de racine s qui contient (u, v) et un autre qui ne contient pas (u, v) .

24.5.3 Reprendre la démonstration du lemme 24.10 en tenant compte des cas où les longueurs de plus court chemin valent ∞ ou $-\infty$.

24.5.4 Soit $G = (S, A)$ un graphe orienté pondéré de sommet origine s , initialisé par SOURCE-UNIQUE-INITIALISATION(G, s). Démontrer que si une séquence d'étapes de relâchement donne à $\pi[s]$ une valeur autre que NIL, alors G contient un circuit de longueur strictement négative.

24.5.5 Soit $G = (S, A)$ un graphe orienté pondéré, sans arc de poids négatif. Soit $s \in S$ le sommet origine et supposons que l'on autorise $\pi[v]$ à être le prédécesseur de v sur *n'importe quel* plus court chemin entre v et l'origine s si $v \in S - \{s\}$ est accessible depuis s , et qu'on lui donne la valeur NIL sinon. Donner un exemple d'un tel graphe et une assignation de valeurs π qui produise un circuit dans G_π . (D'après le lemme 24.16, une telle assignation ne peut pas être produite par une suite d'étapes de relâchement.)

24.5.6 Soit $G = (S, A)$ un graphe orienté pondéré, de fonction de pondération $w : A \rightarrow \mathbf{R}$, et sans circuit de longueur strictement négative. Soit $s \in S$ le sommet origine et supposons que G soit initialisé par SOURCE-UNIQUE-INITIALISATION(G, s). Démontrer que pour tout sommet $v \in S_\pi$, il existe un chemin de s vers v dans G_π et que cette propriété est un invariant pour n'importe quelle séquence de relâchements.

24.5.7 Soit $G = (S, A)$ un graphe orienté pondéré sans circuit de longueur strictement négative. Soit $s \in S$ un sommet origine et supposons que G soit initialisé par SOURCE-UNIQUE-INITIALISATION(G, s). Démontrer qu'il existe une séquence de $|S| - 1$ étapes de relâchement qui produit $d[v] = \delta(s, v)$ pour tout $v \in S$.

24.5.8 Soit G un graphe orienté pondéré arbitraire, comportant un circuit de longueur strictement négative accessible depuis le sommet origine s . Montrer qu'une séquence infinie de relâchements sur les arcs de G peut toujours être construite de manière que chaque relâchement provoque une modification de l'estimation de plus court chemin.

PROBLÈMES

24.1. Amélioration de Yen pour l'algorithme de Bellman-Ford

Supposons que les relâchements d'arc effectués à chaque passage de l'algorithme de Bellman-Ford soient ordonnés comme suit. Avant le premier passage, on considère un ordre linéaire arbitraire $v_1, v_2, \dots, v_{|V|}$ des sommets du graphe $G = (S, A)$. Ensuite, on partitionne l'ensemble d'arcs A en $A_f \cup A_b$, où $A_f = \{(v_i, v_j) \in A : i < j\}$ et $A_b = \{(v_i, v_j) \in A : i > j\}$. (On suppose que G ne contient pas de boucles, de sorte que chaque arc est dans A_f ou dans A_b .) On définit $G_f = (S, A_f)$ et $G_b = (S, A_b)$.

- a. Démontrer que G_f est sans circuit avec le tri topologique $\langle v_1, v_2, \dots, v_{|V|} \rangle$ et que G_b est sans circuit en utilisant le tri topologique $\langle v_{|V|}, v_{|V|-1}, \dots, v_1 \rangle$.

On suppose que chaque passage de l'algorithme de Bellman-Ford est implémenté de la manière suivante. Chaque sommet est visité dans l'ordre $v_1, v_2, \dots, v_{|S|}$, en relâchant les arcs de A_f qui sortent du sommet. Chaque sommet est ensuite visité dans l'ordre $v_{|S|}, v_{|S|-1}, \dots, v_1$, en relâchant les arcs de A_b qui sortent du sommet.

- b. Démontrer qu'avec ce modèle, si G ne contient aucun circuit de longueur strictement négative accessible à partir du sommet origine s , alors après seulement $\lceil |S|/2 \rceil$ passages sur les arcs, $d[v] = \delta(s, v)$ pour tous les sommets $v \in S$.
- c. Ce modèle améliore-t-il le temps d'exécution asymptotique de l'algorithme de Bellman-Ford ?

24.2. Boîtes imbriquées

Une boîte à d dimensions (x_1, x_2, \dots, x_d) est *imbriquée* dans une autre de dimensions (y_1, y_2, \dots, y_d) s'il existe dans $\{1, 2, \dots, d\}$ une permutation π telle que $x_{\pi(1)} < y_1$, $x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d$.

- a. Montrer que la relation d'imbrication est transitive.
- b. Décrire une méthode efficace permettant de déterminer si une boîte de dimension d est imbriquée dans une autre.
- c. Supposons qu'on dispose d'un ensemble de n boîtes de dimension d $\{B_1, B_2, \dots, B_n\}$. Décrire un algorithme efficace permettant de déterminer la plus longue séquence $\langle B_{i_1}, B_{i_2}, \dots, B_{i_k} \rangle$ de boîtes telle que, pour $j = 1, 2, \dots, k - 1$, B_{i_j} soit imbriquée dans $B_{i_{j+1}}$. Exprimer le temps d'exécution de votre algorithme en fonction de n et d .

24.3. Arbitrage

L'arbitrage est l'utilisation du décalage entre les taux de change d'une monnaie pour transformer une unité de monnaie en plus d'une unité de la même monnaie. Par exemple, supposons que 1 dollar U.S. permette d'acheter 0,7 livre britannique, que 1 livre britannique vaille 9,5 francs français, et que 1 franc français vaille 0,16 dollar U.S. Alors, en convertissant les monnaies, un spéculateur peut commencer avec 1 dollar U.S. et acheter $0,7 \times 9,5 \times 0,16 = 1,064$ dollar U.S., soit un profit de 6,4 pour cent.

Supposons qu'on dispose de n monnaies données c_1, c_2, \dots, c_n et d'un tableau $n \times n$ de taux de change R , de manière qu'une unité de la monnaie c_i permette d'acheter $R[i, j]$ unités de la monnaie c_j .

- a. Donner un algorithme efficace permettant de déterminer s'il existe une séquence de monnaies $\langle c_{i_1}, c_{i_2}, \dots, c_{i_k} \rangle$ telle que

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Analyser le temps d'exécution de votre algorithme.

- b. Donner un algorithme efficace permettant d'imprimer une telle séquence si elle existe. Analyser son temps d'exécution.

24.4. Algorithme de Gabow pour plus courts chemins à origine unique

Un algorithme de *changement d'échelle* résout un problème en ne considérant initialement que le bit d'ordre supérieur de chaque valeur d'entrée pertinente (un poids d'arc, par exemple). Puis il affine la solution initiale en regardant les deux bits d'ordre supérieur. Il regarde ainsi progressivement de plus en plus de bits, affinant la solution à chaque fois, jusqu'à ce que tous les bits aient été considérés et que la solution correcte ait été calculée.

Dans ce problème, on examine un algorithme permettant de calculer les plus courts chemins à partir d'une origine unique, en faisant varier l'échelle des poids d'arc. Soit un graphe orienté $G = (S, A)$ dont la fonction de pondération d'arc w est entière et non négative. Soit $W = \max_{(u,v) \in A} \{w(u, v)\}$. Notre but est de développer un algorithme qui s'exécute en $O(A \lg W)$. On suppose que tous les sommets sont accessibles à partir de l'origine.

L'algorithme considère les bits de la représentation binaire des poids d'arc un par un, en partant du bit le plus significatif jusqu'au moins significatif. Plus précisément, soit $k = \lceil \lg(W + 1) \rceil$ le nombre de bits de la représentation binaire de W , et pour $i = 1, 2, \dots, k$, soit $w_i(u, v) = \lfloor w(u, v) / 2^{k-i} \rfloor$. Alors, $w_i(u, v)$ est la version « réduite » de $w(u, v)$ donnée par les i bits les plus significatifs de $w(u, v)$. (Ainsi, $w_k(u, v) = w(u, v)$ pour tout $(u, v) \in A$.) Par exemple, si $k = 5$ et $w(u, v) = 25$, représenté en binaire par $\langle 11001 \rangle$, alors $w_3(u, v) = \langle 110 \rangle = 6$. Autre exemple avec $k = 5$: si $w(u, v) = \langle 00100 \rangle = 4$, alors $w_3(u, v) = \langle 001 \rangle = 1$. Définissons $\delta_i(u, v)$ comme étant la longueur de plus court chemin depuis le sommet u vers le sommet v pour la fonction de pondération w_i . Dans ce cas, $\delta_k(u, v) = \delta(u, v)$ pour tout $u, v \in S$. Pour un sommet origine s donné, l'algorithme de changement d'échelle calcule d'abord les longueurs de plus court chemin $\delta_1(s, v)$ pour tout $v \in S$, puis calcule $\delta_2(s, v)$ pour tout $v \in S$, et ainsi de suite, jusqu'à calculer $\delta_k(s, v)$ pour tout $v \in S$. On suppose que l'on a toujours $|A| \geq |S| - 1$ et l'on verra que le calcul de δ_i à partir de δ_{i-1} demande un temps $O(A)$, de sorte que l'algorithme tout entier prend un temps $O(kA) = O(A \lg W)$.

- a. Supposons que pour tous les sommets $v \in S$, on ait $\delta(s, v) \leq |A|$. Montrer qu'il est possible de calculer $\delta(s, v)$ pour tout $v \in S$ en temps $O(A)$.
- b. Montrer qu'il est possible de calculer $\delta_1(s, v)$ pour tout $v \in S$ en temps $O(A)$.

Intéressons-nous au calcul de δ_i à partir de δ_{i-1} .

- c. Démontrer que pour $i = 2, 3, \dots, k$, on a soit $w_i(u, v) = 2w_{i-1}(u, v)$, soit $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Puis, démontrer que

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |S| - 1$$

pour tout $v \in S$.

- d. On définit pour $i = 2, 3, \dots, k$ et tout $(u, v) \in A$,

$$\widehat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v).$$

Démontrer que pour $i = 2, 3, \dots, k$ et tout couple $u, v \in S$, la valeur « repondérée » $\widehat{w}_i(u, v)$ de l'arc (u, v) est un entier positif.

- e. On définit à présent $\widehat{\delta}_i(s, v)$ comme la longueur de plus court chemin de s vers v pour la fonction de pondération \widehat{w}_i . Démontrer que pour $i = 2, 3, \dots, k$ et tout $v \in S$,

$$\delta_i(s, v) = \widehat{\delta}_i(s, v) + 2\delta_{i-1}(s, v),$$

et que $\widehat{\delta}_i(s, v) \leq |A|$.

- f. Montrer comment calculer $\delta_i(s, v)$ à partir de $\delta_{i-1}(s, v)$ pour tout $v \in S$ en temps $O(A)$, et conclure que $\delta(s, v)$ peut être calculé pour tout $v \in S$ en temps $O(A \lg W)$.

24.5. Algorithme de Karp pour circuit de poids moyen minimal

Soit $G = (S, A)$ un graphe orienté, de fonction de pondération $w : A \rightarrow \mathbf{R}$, et soit

$n = |S|$. On définit le **poids moyen** d'un circuit $c = \langle e_1, e_2, \dots, e_k \rangle$ par

$$\mu(c) = \frac{1}{k} \sum_{i=1}^k w(e_i).$$

Soit $\mu^* = \min_c \mu(c)$, où c décrit l'ensemble des circuits de G . Un circuit c pour lequel $\mu(c) = \mu^*$ est appelé **circuit de poids moyen minimal**. Nous allons étudier ici un algorithme efficace permettant de calculer μ^* .

Supposons sans perte de généralité que chaque sommet $v \in S$ est accessible à partir d'un sommet origine $s \in S$. Soit $\delta(s, v)$ la longueur d'un plus court chemin de s vers v , et soit $\delta_k(s, v)$ la longueur d'un plus court chemin de s vers v constitué de k arcs *exactement*. S'il n'existe pas de chemin de s à v composé de k arcs exactement, alors $\delta_k(s, v) = \infty$.

- a. Montrer que si $\mu^* = 0$, alors G ne contient aucun circuit de longueur strictement négative, et $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ pour tous les sommets $v \in S$.
- b. Montrer que si $\mu^* = 0$, alors

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0$$

pour tous les sommets $v \in S$. (*conseil* : Utiliser les deux propriétés de la partie (a).)

- c. Soit c un circuit de longueur 0, et soient u et v deux sommets quelconques de c . On suppose que $\mu^* = 0$ et que la longueur du chemin de u vers v vaut x le long du circuit. Démontrer que $\delta(s, v) = \delta(s, u) + x$. (*Conseil* : La longueur du chemin de v vers u le long du circuit est $-x$.)
- d. Montrer que si $\mu^* = 0$, il existe un sommet v sur chaque circuit de poids moyen minimal tel que

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

(*Conseil* : Montrer qu'un plus court chemin vers un sommet quelconque du circuit de poids moyen minimal peut être étendu le long du circuit pour créer un plus court chemin vers le sommet suivant dans le circuit.)

- e. Montrer que si $\mu^* = 0$, alors

$$\min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

- f. Montrer que si l'on ajoute une constante t au poids de chaque arc de G , alors μ^* est augmenté de t . Se servir de cette propriété pour montrer que

$$\mu^* = \min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}.$$

- g. Donner un algorithme à temps $O(SA)$ permettant de calculer μ^* .

24.6. Plus courts chemins bitoniques

Une séquence est *bitonique* si elle est monotone croissante puis monotone décroissante (moyennant un éventuel décalage circulaire). Ainsi, les suites $\langle 1, 4, 6, 8, 3, -2 \rangle$, $\langle 9, 2, -4, -10, -5 \rangle$ et $\langle 1, 2, 3, 4 \rangle$ sont bitoniques, mais $\langle 1, 3, 12, 4, 2, 10 \rangle$ n'est pas bitonique. (Voir le chapitre 27 pour une étude des trieuses bitoniques, et voir le problème 15.1 pour le problème euclidien bitonique du voyageur de commerce.)

Soit un graphe orienté $G = (S, A)$ de fonction de pondération $w : A \rightarrow \mathbf{R}$, dans lequel on veut trouver les plus courts chemins à origine unique qui partent d'un sommet origine s . On dispose d'une information supplémentaire : pour chaque sommet $v \in S$, les poids des arcs le long d'un plus court chemin de s à v forment une séquence bitonique.

Donner l'algorithme le plus efficace que vous puissiez trouver pour ce problème et analyser son temps d'exécution.

NOTES

L'algorithme de Dijkstra [75] a été trouvé en 1959, mais il ne mentionnait pas de file de priorités. L'algorithme de Bellman-Ford est basé sur des algorithmes séparés de Bellman [35] et Ford [93]. Bellman décrit la relation entre plus courts chemins et contraintes de potentiel. Lawler [196] décrit l'algorithme à temps linéaire pour les plus courts chemins d'un graphe orienté sans circuit, qu'il considère comme faisant partie du folklore.

Quand les poids d'arc sont relativement petits, on peut utiliser des algorithmes plus efficaces pour résoudre le problème des plus courts chemins à origine unique. La suite de valeurs retournée par les appels EXTRAIRE-MIN de l'algorithme de Dijkstra croît de façon monotone avec le temps. Comme mentionné dans les notes du chapitre 6, en pareil cas il existe plusieurs structures de données permettant d'implémenter les diverses opérations de file de priorité plus efficacement que ne le permettent les tas binaires ou les tas de Fibonacci. Ahuja, Mehlhorn, Orlin et Tarjan [8] donnent un algorithme qui s'exécute en temps $O(A + S\sqrt{\lg W})$ sur les graphes dont les poids d'arc sont positifs ou nuls, où W est le poids maximal sur tous les arcs du graphe. Les meilleures bornes sont dues à Thorup [299], qui donne un algorithme à temps $O(A \lg \lg S)$, et à Raman, qui donne un algorithme à temps $O(A + S \min \{(\lg S)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\})$. Ces deux algorithmes utilisent une quantité de mémoire qui dépend de la taille du mot sur la machine sous-jacente. Le volume de mémoire consommé peut être non borné par rapport à la taille de l'entrée, mais un hachage randomisé permet de l'amener à être linéaire par rapport à la taille de l'entrée.

Pour les graphes non orientés à poids entiers, Thorup [298] donne un algorithme à temps $O(S + A)$ pour le calcul des plus courtes chaînes à origine unique. Par comparaison avec les algorithmes cités au paragraphe précédent, cet algorithme n'est pas une implémentation de celui de Dijkstra, vu que la séquence de valeurs retournée par les appels EXTRAIRE-MIN n'est pas monotone croissante avec le temps.

Pour les graphes contenant des arcs de poids négatif, un algorithme dû à Gabow et Tarjan [104] s'exécute en $O(\sqrt{SA} \lg(SW))$, et un algorithme dû à Goldberg [118] s'exécute en temps $O(\sqrt{SA} \lg W)$, où $W = \max_{(u,v) \in A} \{|w(u,v)|\}$.

Cherkassky, Goldberg et Radzik [57] ont fait des expériences exhaustives de comparaison entre algorithmes de plus courts chemins.

Chapitre 25

Plus courts chemins pour tout couple de sommets

Dans ce chapitre, nous nous intéresserons à la recherche des plus courts chemins entre tous les couples de sommets d'un graphe. Ce problème peut intervenir pendant l'élaboration d'une table des distances entre tous les couples de villes dans un atlas routier. Comme dans le chapitre 24, on dispose en entrée d'un graphe orienté pondéré $G = (S, A)$ ayant la fonction de pondération $w : A \rightarrow \mathbf{R}$ qui associe à chaque arc un poids à valeur réelle. On souhaite déterminer, pour tout couple de sommets $u, v \in S$, un plus court chemin (de longueur minimale) de u à v , où la longueur d'un chemin est la somme des poids des arcs qui le constituent. En général, les résultats seront disposés sous forme d'un tableau : l'intersection de la ligne u et de la colonne v contiendra la longueur d'un plus court chemin entre u et v .

On peut résoudre un problème de plus court chemin portant sur tous les couples de sommets en exécutant $|S|$ fois un algorithme de plus court chemin à origine unique, en prenant à chaque fois comme origine un nouveau sommet. Si tous les poids d'arc sont positifs ou nuls, on peut utiliser l'algorithme de Dijkstra. Si la file de priorité min est implémentée sous la forme d'un tableau linéaire, le temps d'exécution est en $O(S^3 + SA) = O(S^3)$. L'implémentation par tas min binaire de la file de priorité donne un temps d'exécution en $O(SA \lg S)$, ce qui est une amélioration si le graphe est peu dense. On peut aussi implémenter la file de priorité min à l'aide d'un tas de Fibonacci, ce qui engendre un temps d'exécution en $O(S^2 \lg S + SA)$.

Si l'on autorise les arcs de poids négatif, l'algorithme de Dijkstra n'est plus utilisable. Il faut alors exécuter l'algorithme de Bellman-Ford, qui est plus lent, une fois

depuis chaque sommet. Le temps d'exécution résultant est en $O(S^2A)$, ce qui donne $O(S^4)$ sur un graphe dense. Dans ce chapitre, nous verrons qu'il est possible de faire mieux. Nous étudierons aussi la relation entre le problème des plus courts chemins pour tout couple de sommet et la multiplication des matrices, ainsi que sa structure algébrique.

Contrairement aux algorithmes à origine unique, qui supposent que le graphe est représenté par une liste d'adjacences, la plupart des algorithmes de ce chapitre utilisent une représentation par matrice d'adjacences. (L'algorithme de Johnson pour les graphes peu denses utilise des listes d'adjacences.) Pour des raisons de commodités, nous supposons que les sommets sont numérotés $1, 2, \dots, |S|$, de sorte que l'entrée est une matrice W de type $n \times n$ qui représente les poids d'arc d'un graphe orienté $G = (S, A)$ à n sommets. Autrement dit, $W = (w_{ij})$, où

$$w_{ij} = \begin{cases} 0 & \text{si } i = j, \\ \text{le poids de l'arc } (i, j) & \text{si } i \neq j \text{ et } (i, j) \in A, \\ \infty & \text{si } i \neq j \text{ et } (i, j) \notin A. \end{cases} \quad (25.1)$$

Les arcs de poids négatif sont autorisés, mais on suppose pour l'instant que le graphe en entrée ne contient aucun circuit de longueur strictement négative.

La sortie tabulaire produite par les algorithmes de plus court chemin toutes paires qui sont présentés dans ce chapitre est une matrice $n \times n$ notée $D = (d_{ij})$, où la composante d_{ij} contient la longueur d'un plus court chemin reliant le sommet i au sommet j . Autrement dit, si l'on appelle $\delta(i, j)$ la longueur de plus court chemin du sommet i au sommet j (comme au chapitre 24), alors $d_{ij} = \delta(i, j)$ à la fin de l'exécution.

Pour rechercher les plus courts chemins entre tous les couples de sommets à partir d'une matrice d'adjacences donnée, il faut calculer non seulement les longueurs des plus courts chemins mais aussi une **matrice de liaison** $\Pi = (\pi_{ij})$, où π_{ij} vaut NIL si $i = j$ ou s'il n'existe aucun chemin de i vers j ; autrement, π_{ij} est le prédécesseur de j sur un plus court chemin issu de i . De même que le sous-graphe de liaison G_π du chapitre 24 est une arborescence de plus courts chemins pour un sommet origine donné, le sous-graphe induit par la i ème ligne de la matrice Π sera une arborescence de plus courts chemins ayant pour racine i . Pour chaque sommet $i \in S$, on définit le **sous-graphe de liaison** de G pour i par $G_{\pi, i} = (S_{\pi, i}, A_{\pi, i})$, où

$$S_{\pi, i} = \{j \in S : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

et

$$A_{\pi, i} = \{(\pi_{ij}, j) : j \in S_{\pi, i} - \{i\}\}.$$

Si $G_{\pi, i}$ est une arborescence de plus courts chemins, la procédure ci-après, qui est une version modifiée de la procédure IMPRIMER-CHEMIN du chapitre 24, imprime un plus court chemin reliant le sommet i au sommet j .

```

IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES( $\Pi, i, j$ )
1  si  $i = j$ 
2    alors imprimer  $i$ 
3    sinon si  $\pi_{ij} = \text{NIL}$ 
4        alors imprimer « il n'existe aucun chemin de »  $i$  « à »  $j$ 
5        sinon IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES( $\Pi, i, \pi_{ij}$ )
6            imprimer  $j$ 

```

Afin de mieux faire ressortir les caractéristiques principales des algorithmes toutes paires, nous n'étudierons pas la création et les propriétés des matrices de liaison de manière aussi détaillée que pour les sous-graphes de liaison du chapitre 24. Les notions élémentaires seront traitées par certains exercices.

a) Structure du chapitre

La section 25.1 présente un algorithme de programmation dynamique, qui se fonde sur la multiplication des matrices, pour résoudre le problème des plus courts chemins pour tout couple de sommets. Grâce à la technique des « élévations au carré », on peut faire en sorte que cet algorithme s'exécute en $\Theta(S^3 \lg S)$. Un autre algorithme de programmation dynamique, l'algorithme de Floyd-Warshall, est donné à la section 25.2. Celui-là s'exécute en $\Theta(S^3)$. La section 25.2 traite aussi le problème de la fermeture transitive d'un graphe orienté, qui est lié à celui des plus courts chemins pour tous couples de sommets. Enfin, l'algorithme de Johnson est présenté à la section 25.3. Contrairement aux autres algorithmes de ce chapitre, l'algorithme de Johnson utilise pour un graphe la représentation par listes d'adjacences. Il trouve les plus courts chemins entre tous les couples de sommets en $O(S^2 \lg S + SA)$, ce qui en fait un bon algorithme pour les grand graphes peu denses.

Avant de continuer, il est nécessaire de définir quelques conventions concernant les représentations par matrices d'adjacences. Premièrement, nous supposons généralement que le graphe d'entrée $G = (S, A)$ comporte n sommets, c'est-à-dire que $|S| = n$. Deuxièmement, les matrices seront notées par des majuscules, comme W ou D , et leurs composantes par des minuscules indicées, comme w_{ij} ou d_{ij} . Certaines matrices porteront des exposants entre parenthèses, comme dans $D^{(m)} = (d_{ij}^{(m)})$, pour indiquer des itérations. Enfin, pour une matrice M donnée $n \times n$, nous supposons que la valeur de n est stockée dans l'attribut *lignes*[M].

25.1 PLUS COURTS CHEMINS ET MULTIPLICATION DE MATRICES

Cette section présente un algorithme de programmation dynamique pour le problème des plus courts chemins pour tout couple de sommets sur un graphe orienté $G = (S, A)$. Chaque exécution de la boucle principale du programme dynamique

effectue une opération très similaire à un produit de matrices, et fait ressembler l'algorithme à une multiplication répétée de matrices. Nous allons commencer par développer un algorithme en $\Theta(S^4)$, puis nous améliorerons son temps d'exécution pour le faire descendre à $\Theta(S^3 \lg S)$.

Avant d'aller plus loin, récapitulons brièvement les étapes suggérées au chapitre 15 pour l'élaboration d'un algorithme de programmation dynamique.

- 1) Caractérisation de la structure d'une solution optimale.
- 2) Définition récursive de la valeur d'une solution optimale.
- 3) Calcul de la valeur d'une solution optimale de façon ascendante.

(La quatrième étape, construction d'une solution optimale à partir des informations calculées, sera traitée en exercice.)

b) La structure d'un plus court chemin

On commence par caractériser la structure d'une solution optimale. Concernant les plus courts chemins pour tous les couples de sommets d'un graphe $G = (S, A)$, nous avons démontré (Lemme 24.1) que tous les sous-chemins d'un plus court chemin sont eux-mêmes des plus courts chemins. On suppose que le graphe est représenté par une matrice d'adjacences $W = (w_{ij})$. On considère un plus court chemin p du sommet i au sommet j , et on suppose que p contient au plus m arcs. En supposant qu'il n'existe aucun circuit de longueur strictement négative, m est fini. Si $i = j$, alors p a une longueur égale à 0 et n'a aucun arc. Si les sommets i et j sont distincts, on décompose le chemin p en $i \xrightarrow{p'} k \rightarrow j$, où le chemin p' contient maintenant au plus $m - 1$ arcs. D'après le lemme 24.1 p' est un plus court chemin de i vers k , et donc $\delta(i, j) = \delta(i, k) + w_{kj}$.

c) Une solution récursive au problème des plus courts chemins pour tout couple de sommets

Soit $d_{ij}^{(m)}$ la longueur minimale d'un chemin d'au plus m arcs du sommet i au sommet j . Pour $m = 0$, il existe un plus court chemin sans arc de i vers j si et seulement si $i = j$. Donc,

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{si } i = j, \\ \infty & \text{si } i \neq j. \end{cases}$$

Pour $m \geq 1$, on calcule $d_{ij}^{(m)}$ comme étant le minimum de $d_{ij}^{(m-1)}$ (poids du plus court chemin de i à j constitué d'au plus $m - 1$ arcs) et de la longueur minimale d'un quelconque chemin d'au plus m arcs de i à j , obtenu après examen de tous les prédécesseurs potentiels k de j . On définit donc récursivement

$$\begin{aligned} d_{ij}^{(m)} &= \min \left(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + w_{kj}\} \right) \\ &= \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + w_{kj}\}. \end{aligned} \quad (25.2)$$

La dernière égalité vient du fait que $w_{jj} = 0$ pour tout j .

Quels sont les longueurs réelles de plus court chemin $\delta(i, j)$? Si le graphe ne contient aucun circuit de longueur négative, alors pour toute paire de sommets i et j tels que $\delta(i, j) < \infty$, il existe un plus court chemin de i à j qui est élémentaire et qui contient donc au plus $n - 1$ arcs. Un chemin du sommet i au sommet j de plus de $n - 1$ arcs ne peut pas avoir une longueur inférieure à celle d'un plus court chemin de i à j . Les longueurs de plus court chemin réelles sont donc données par

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \dots \quad (25.3)$$

d) Calcul ascendant des longueurs de plus court chemin

En prenant en entrée la matrice $W = (w_{ij})$, on calcule maintenant une série de matrices $D^{(1)}, D^{(2)}, \dots, D^{(n-1)}$ où, pour $m = 1, 2, \dots, n - 1$, on a $D^{(m)} = (d_{ij}^{(m)})$. La dernière matrice, $D^{(n-1)}$, contient les longueurs de plus court chemin réelles. Notons que, comme $d_{ij}^{(1)} = w_{ij}$ pour tous les sommets $i, j \in S$, on a $D^{(1)} = W$.

Le cœur de l'algorithme est constitué de la procédure suivante qui, étant données les matrices $D^{(m-1)}$ et W , retourne la matrice $D^{(m)}$. Autrement dit, elle étend d'un arc supplémentaire les plus courts chemins calculés jusqu'à présent.

EXTENSION-PLUS-COURTS-CHEMINS(D, W)

```

1   $n \leftarrow \text{lignes}[D]$ 
2  soit  $D' = (d'_{ij})$  une matrice  $n \times n$ 
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire pour  $j \leftarrow 1$  à  $n$ 
5          faire  $d'_{ij} \leftarrow \infty$ 
6              pour  $k \leftarrow 1$  à  $n$ 
7                  faire  $d'_{ij} \leftarrow \min(d'_{ij}, d_{ik} + w_{kj})$ 
8  retourner  $D'$ 
```

La procédure calcule une matrice $D' = (d'_{ij})$, qu'elle retourne à la fin. Pour cela, elle résout l'équation (25.2) pour tout i et j , en prenant D pour $D^{(m-1)}$ et D' pour $D^{(m)}$. (Elle ne fait pas intervenir les exposants, de façon que les matrices en entrée et en sortie soient indépendantes de m .) Son temps d'exécution en $\Theta(n^3)$ est dû aux trois boucles **pour** imbriquées.

On peut voir à présent la relation avec la multiplication des matrices. Supposons qu'on veuille calculer le produit $C = A \cdot B$ de deux matrices A et B de taille $n \times n$. Pour $i, j = 1, 2, \dots, n$, on calcule

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj} \quad (25.4)$$

Notons que, si l'on effectue les substitutions

$$\begin{aligned} d^{(m-1)} &\rightarrow a, \\ w &\rightarrow b, \\ d^{(m)} &\rightarrow c, \\ \min &\rightarrow +, \\ + &\rightarrow \cdot \end{aligned}$$

dans l'équation (25.2), on obtient l'équation (25.4). Donc, si ces modifications sont répercutées dans EXTENSION-PLUS-COURTS-CHEMINS et que ∞ (l'élément neutre pour min) est remplacé par 0 (l'élément neutre pour +), on obtient la procédure simple qui multiplie deux matrices en $\Theta(n^3)$:

MULTIPLIER-MATRICES(A, B)

```

1   $n \leftarrow \text{lignes}[A]$ 
2  soit  $C$  une matrice  $n \times n$ 
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire pour  $j \leftarrow 1$  à  $n$ 
5          faire  $c_{ij} \leftarrow 0$ 
6              pour  $k \leftarrow 1$  à  $n$ 
7                  faire  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
8  retourner  $C$ 
```

Pour en revenir au problème des plus courts chemins entre tous les couples de sommets, on calcule les longueurs de plus court chemin en allongeant les plus courts chemins arc par arc. En notant $A \cdot B$ la matrice « produit » retournée par EXTENSION-PLUS-COURTS-CHEMINS(A, B), on calcule la séquence des $n - 1$ matrices

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot W = W, \\ D^{(2)} &= D^{(1)} \cdot W = W^2, \\ D^{(3)} &= D^{(2)} \cdot W = W^3, \\ &\vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot W = W^{n-1}. \end{aligned}$$

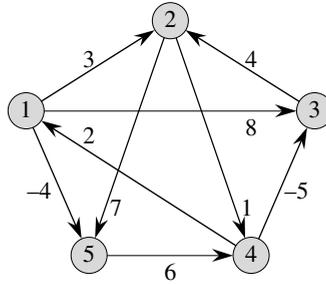
Comme nous l'avons montré plus haut, la matrice $D^{(n-1)} = W^{n-1}$ contient les poids de plus court chemin. La procédure suivante calcule cette séquence en un temps $\Theta(n^4)$.

PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI(W)

```

1   $n \leftarrow \text{lignes}[W]$ 
2   $D^{(1)} \leftarrow W$ 
3  pour  $m \leftarrow 2$  à  $n - 1$ 
4      faire  $D^{(m)} \leftarrow \text{EXTENSION-PLUS-COURTS-CHEMINS}(D^{(m-1)}, W)$ 
5  retourner  $D^{(n-1)}$ 
```

La figure 25.1 montre un graphe et les matrices $D^{(m)}$ calculées par la procédure PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI.



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figure 25.1 Un graphe orienté, et la séquence des matrices $D^{(m)}$ calculées par PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI. Le lecteur pourra vérifier que $D^{(5)} = D^{(4)} \cdot W$ est égale à $D^{(4)}$, d'où $D^{(m)} = D^{(4)}$ pour tout $m \geq 4$.

e) Amélioration du temps d'exécution

Cependant, notre but n'est pas de calculer *toutes* les matrices $D^{(m)}$: seule la matrice $D^{(n-1)}$ nous intéresse. Rappelons-nous que, en l'absence de circuits de longueur strictement négative, l'équation (25.3) implique $D^{(m)} = D^{(n-1)}$ pour tout entier $m \geq n - 1$. De même que la multiplication matricielle classique est associative, de même la multiplication matricielle définie par la procédure EXTENSION-PLUS-COURTS-CHEMINS est associative (voir exercice 25.1.4). On peut calculer $D^{(n-1)}$ avec seulement $\lceil \lg(n-1) \rceil$ produits de matrices, en calculant la séquence

$$\begin{aligned} D^{(1)} &= W, \\ D^{(2)} &= W^2 = W \cdot W, \\ D^{(4)} &= W^4 = W^2 \cdot W^2, \\ D^{(8)} &= W^8 = W^4 \cdot W^4, \\ &\vdots \\ D^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}. \end{aligned}$$

Comme $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, le produit final $D^{(2^{\lceil \lg(n-1) \rceil})}$ est égal à $D^{(n-1)}$.

La procédure suivante calcule la séquence de matrices précédente en utilisant cette technique d'*élévations au carré répétées*.

PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ(W)

```

1   $n \leftarrow \text{lignes}[W]$ 
2   $D^{(1)} \leftarrow W$ 
3   $m \leftarrow 1$ 
4  tant que  $m < n - 1$ 
5      faire  $D^{(2m)} \leftarrow \text{EXTENSION-PLUS-COURTS-CHEMINS}(D^{(m)}, D^{(m)})$ 
6           $m \leftarrow 2m$ 
7  retourner  $D^{(m)}$ 

```

A chaque itération de la boucle **tant que** des lignes 4–6, on calcule $D^{(2m)} = (D^{(m)})^2$, en commençant avec $m = 1$. A la fin de chaque itération, la valeur de m est doublée. La dernière itération calcule $D^{(n-1)}$ en calculant en réalité $D^{(2m)}$ pour un certain $n - 1 \leq 2m < 2n - 2$. D'après l'équation (25.3), $D^{(2m)} = D^{(n-1)}$. A l'évaluation suivante du test de la ligne 4, m a été doublé, de sorte que maintenant $m \geq n - 1$; le test échoue, et la procédure retourne la dernière matrice qu'elle a calculée.

Le temps d'exécution de PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ est en $\Theta(n^3 \lg n)$ puisque chacun des $\lceil \lg(n - 1) \rceil$ produits de matrices consomme un temps en $\Theta(n^3)$. Observez que le code est compact, ne contenant aucune structure de données élaborée. La constante cachée dans la notation Θ est donc petite.

Exercices

25.1.1 Exécuter PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI sur le graphe orienté pondéré de la figure 25.2, en montrant les matrices résultant de chaque itération de la boucle. Faire de même avec PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ.

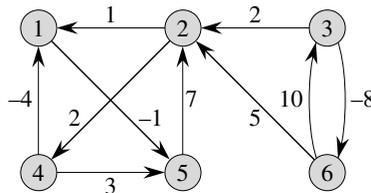


Figure 25.2 Un graphe orienté pondéré servant de support aux exercices 25.1.1, 25.2.1 et 25.3.1.

25.1.2 Pourquoi faut-il que $w_{ii} = 0$ pour tout $1 \leq i \leq n$?

25.1.3 À quoi correspond la matrice

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

utilisée dans les algorithmes de plus court chemin, dans une multiplication classique de matrices ?

25.1.4 Montrer que la multiplication de matrices définie par EXTENSION-PLUS-COURTS-CHEMINS est associative.

25.1.5 Montrer comment exprimer le problème du plus court chemin à origine unique comme produit d'une matrice et d'un vecteur. Expliquer en quoi l'évaluation de ce produit correspond à un algorithme du genre Bellman-Ford (voir section 24.1).

25.1.6 Supposons qu'on veuille aussi calculer les sommets situés sur les plus courts chemins dans les algorithmes de cette section. Montrer comment calculer en temps $O(n^3)$ la matrice de liaison Π à partir de la matrice complétée D des longueurs de plus court chemin.

25.1.7 Les sommets des plus courts chemins peuvent aussi être calculés en même temps que les longueurs des plus courts chemins. Définissons $\pi_{ij}^{(m)}$ comme le prédécesseur du sommet j sur un chemin de longueurs minimale entre i et j contenant au plus m arcs. Modifier EXTENSION-PLUS-COURTS-CHEMINS et PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI pour calculer les matrices $\Pi^{(1)}, \Pi^{(2)}, \dots, \Pi^{(n-1)}$ en même temps que les matrices $D^{(1)}, D^{(2)}, \dots, D^{(n-1)}$.

25.1.8 La procédure PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ, telle qu'elle est écrite, impose le stockage de $\lceil \lg(n-1) \rceil$ matrices, contenant chacune n^2 composantes, pour un espace total $\Theta(n^2 \lg n)$. Modifier la procédure pour qu'elle ne consomme qu'un espace en $\Theta(n^2)$, en lui faisant utiliser seulement deux matrices $n \times n$.

25.1.9 Modifier la procédure PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ de façon qu'elle détecte la présence d'un circuit de longueur strictement négative.

25.1.10 Donner un algorithme efficace pour trouver la longueur (nombre d'arcs) d'un circuit de longueur strictement négative de longueur minimale dans un graphe.

25.2 L'ALGORITHME DE FLOYD-WARSHALL

Dans cette section, nous utiliserons une autre méthode de programmation dynamique pour résoudre le problème des plus courts chemins pour tout couple de sommets d'un graphe orienté $G = (S, A)$. L'algorithme résultant, connu sous le nom d'*algorithme*

de *Floyd-Warshall*, s'exécute en $\Theta(S^3)$. Comme précédemment, il peut y avoir des arcs de poids négatif, mais nous supposons qu'il n'existe aucun circuit de longueur strictement négative. Comme dans la section 25.1, nous suivrons les étapes de la programmation dynamique pour développer l'algorithme. Après avoir étudié l'algorithme résultant, nous présenterons une méthode similaire permettant de trouver la fermeture transitive d'un graphe orienté.

a) Structure d'un plus court chemin

Pour l'algorithme de Floyd-Warshall, la structure d'un plus court chemin est caractérisée différemment que pour les algorithmes toutes-paires basés sur la multiplication des matrices. L'algorithme considère les sommets « intermédiaires » d'un plus court chemin ; un sommet *intermédiaire* d'un chemin simple $p = \langle v_1, v_2, \dots, v_l \rangle$ est un sommet de p autre que v_1 ou v_l , autrement dit un sommet appartenant à l'ensemble $\{v_2, v_3, \dots, v_{l-1}\}$.

L'algorithme de Floyd-Warshall s'appuie sur l'observation suivante : si l'on appelle $S = \{1, 2, \dots, n\}$ les sommets de G , on considère un sous-ensemble $\{1, 2, \dots, k\}$ de sommets pour un certain k . Pour un couple quelconque de sommets $i, j \in S$, on considère tous les chemins de i à j dont les sommets intermédiaires appartiennent tous à $\{1, 2, \dots, k\}$, et on note p un chemin de longueur minimale parmi eux. (Le chemin p est élémentaire.) L'algorithme de Floyd-Warshall exploite une relation entre le chemin p et les plus courts chemins de i vers j dont tous les sommets intermédiaires sont dans $\{1, 2, \dots, k-1\}$. La relation dépend de ce que k est ou n'est pas un sommet intermédiaire de p .

- Si k n'est pas un sommet intermédiaire du chemin p , alors tous les sommets intermédiaires de p se trouvent dans l'ensemble $\{1, 2, \dots, k-1\}$. Donc, un plus court chemin du sommet i au sommet j ayant tous les sommets intermédiaires dans l'ensemble $\{1, 2, \dots, k-1\}$ est aussi un plus court chemin de i vers j ayant tous les sommets intermédiaires dans l'ensemble $\{1, 2, \dots, k\}$.
- Si k est un sommet intermédiaire du chemin p , alors on divise p en $i \xrightarrow{p_1} k \xrightarrow{p_2} j$ comme illustré à la figure 25.3. D'après le lemme 24.1, p_1 est un plus court chemin de i vers k , tous les sommets intermédiaires appartenant à l'ensemble $\{1, 2, \dots, k\}$. Comme le sommet k n'est pas un sommet intermédiaire de p_1 , on voit que p_1 est un plus court chemin de i vers k , tous les sommets intermédiaires étant pris dans l'ensemble $\{1, 2, \dots, k-1\}$. De même, p_2 est un plus court chemin du sommet k vers le sommet j , tous les sommets intermédiaires étant pris dans $\{1, 2, \dots, k-1\}$.

b) Une solution récursive

En se basant sur les observations précédentes, on définit une formulation récursive des estimations de plus court chemin, différente de celle de la section 25.1. Soit $d_{ij}^{(k)}$ le poids d'un plus court chemin du sommet i au sommet j dont tous les sommets intermédiaires sont dans l'ensemble $\{1, 2, \dots, k\}$. Pour $k = 0$, un chemin de i à j

tous les sommets intermédiaires de $\{1,2,\dots,k-1\}$ tous les sommets intermédiaires de $\{1,2,\dots,k-1\}$

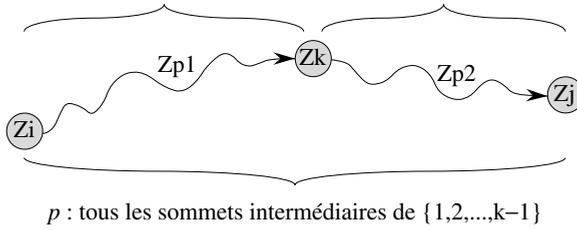


Figure 25.3 Le chemin p est un plus court chemin du sommet i au sommet j , et k est le dernier sommet intermédiaire de p . Tous les sommets intermédiaires du chemin p_1 , lequel est la portion de chemin p du sommet i au sommet k , se trouvent dans l'ensemble $\{1, 2, \dots, k - 1\}$. Idem pour le chemin p_2 menant du sommet k au sommet j .

sans sommet intermédiaire de rang supérieur à 0 ne possède en réalité aucun sommet intermédiaire. Il est constitué d'au plus un arc, et on a donc $d_{ij}^{(0)} = w_{ij}$. Il en résulte la définition récursive que voici

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{si } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{si } k \geq 1. \end{cases} \quad (25.5)$$

La matrice $D^{(n)} = (d_{ij}^{(n)})$ donne le résultat final ($d_{ij}^{(n)} = \delta(i, j)$ pour tout $i, j \in S$); en effet, quel que soit le chemin, tous les sommets intermédiaires appartiennent à l'ensemble $\{1, 2, \dots, n\}$.

c) Calcul ascendant des poids de plus court chemin

Basée sur la récurrence (25.5), la procédure ascendante suivante permet de calculer les valeurs $d_{ij}^{(k)}$ par ordre de valeurs de k croissantes. Elle prend en entrée une matrice $n \times n$, notée W et définie comme dans l'équation (25.1). La procédure retourne la matrice $D^{(n)}$ des longueurs de plus court chemin.

```

FLOYD-WARSHALL(W)
1  n ← lignes[W]
2  D(0) ← W
3  pour k ← 1 à n
4      faire pour i ← 1 à n
5          faire pour j ← 1 à n
6              faire dij(k) ← min(dij(k-1), dik(k-1) + dkj(k-1))
7  retourner D(n)
    
```

La figure 25.4 donne les matrices $D^{(k)}$ calculées par l'algorithme de Floyd-Warshall pour le graphe de la figure 25.1.

Le temps d'exécution de l'algorithme de Floyd-Warshall est déterminé par les trois boucles **pour** imbriquées des lignes 3-6. Chaque exécution de la ligne 6 prenant un

$$\begin{array}{l}
D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}
\end{array}$$

Figure 25.4 La séquence des matrices $D^{(k)}$ et $\Pi^{(k)}$ calculées par l'algorithme de Floyd-Warshall pour le graphe de la figure 25.1.

temps $O(1)$, l'algorithme s'exécute donc en $\Theta(n^3)$. Comme dans le dernier algorithme de la section 25.1, le code est compact et sans structure de données élaborée ; de ce fait, la constante cachée dans la notation Θ est petite. L'algorithme de Floyd-Warshall est donc tout à fait intéressant, y compris pour les graphes de taille modérée.

d) Construction d'un plus court chemin

Il existe de nombreuses méthodes différentes pour construire les plus courts chemins avec l'algorithme de Floyd-Warshall. On peut par exemple calculer la matrice D des longueurs de plus court chemin, puis construire la matrice de liaison Π à partir de la matrice D . Cette méthode peut être implémentée de manière à s'exécuter en $O(n^3)$ (exercice 25.1.6). Connaissant la matrice de liaison Π , on peut utiliser la procédure IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES pour imprimer les sommets d'un plus court chemin donné.

Il est possible de calculer « en ligne » la matrice de liaison Π , en même temps que l'algorithme de Floyd-Warshall calcule les matrices $D^{(k)}$. Plus précisément, on calcule une séquence de matrices $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$, où $\Pi = \Pi^{(n)}$ et $\pi_{ij}^{(k)}$ est le prédécesseur du sommet j sur un plus court chemin partant du sommet i et dont tous les sommets intermédiaires sont dans $\{1, 2, \dots, k\}$.

On peut donner une formulation récursive de $\pi_{ij}^{(k)}$. Pour $k = 0$, un plus court chemin de i vers j ne possède aucun sommet intermédiaire. Donc,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{si } i = j \text{ ou } w_{ij} = \infty, \\ i & \text{si } i \neq j \text{ et } w_{ij} < \infty. \end{cases} \quad (25.6)$$

Pour $k \geq 1$, si l'on prend le chemin $i \rightsquigarrow k \rightsquigarrow j$ où $k \neq j$, alors le prédécesseur de j que nous choisissons est le même que celui que nous avons choisi sur un plus court chemin issu de k dont tous les sommets intermédiaires se trouvent dans $\{1, 2, \dots, k-1\}$. Sinon, on prend le même prédécesseur de j que celui que nous avons choisi sur un plus court chemin partant de i dont tous les sommets intermédiaires sont dans l'ensemble $\{1, 2, \dots, k-1\}$. Formellement, pour $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{si } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{si } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \quad (25.7)$$

L'incorporation des calculs des matrices $\Pi^{(k)}$ dans la procédure FLOYD-WARSHALL est laissé en exercice (voir exercice 25.2.3). La figure 25.4 montre la séquence des matrices $\Pi^{(k)}$ calculées par l'algorithme résultant, à partir du graphe de la figure 25.1. L'exercice demande également d'effectuer le travail plus délicat consistant à démontrer que le sous-graphe de liaison $G_{\pi,i}$ est une arborescence de plus courts chemins de racine i . L'exercice 25.2.7 fournit un autre moyen de reconstruire les plus courts chemins.

e) Fermeture transitive d'un graphe orienté

Étant donné un graphe orienté $G = (S, A)$ où $S = \{1, 2, \dots, n\}$, on désire savoir s'il existe un chemin dans G de i vers j pour tout couple de sommets $i, j \in S$. La **fermeture transitive** de G est définie par le graphe $G^* = (S, A^*)$, où

$$A^* = \{(i, j) : \text{il existe un chemin menant du sommet } i \text{ au sommet } j \text{ dans } G\}.$$

On peut calculer la fermeture transitive d'un graphe en $\Theta(n^3)$, en affectant à chaque arc de A un poids égal à 1 puis et en exécutant l'algorithme de Floyd-Warshall. S'il existe un chemin de i à j , on obtient $d_{ij} < n$. Sinon, on obtient $d_{ij} = \infty$.

Il existe un autre moyen, similaire, de calculer la fermeture transitive de G en $\Theta(n^3)$, qui peut permettre en pratique d'économiser du temps et de l'espace. Cette méthode met en jeu la substitution des opérations logiques \vee (OU logique) et \wedge (ET logique) aux opérations arithmétiques \min et $+$ dans l'algorithme de Floyd-Warshall. Pour $i, j, k = 1, 2, \dots, n$, on définit $t_{ij}^{(k)}$ comme valant 1 s'il existe un chemin, dans le graphe G , du sommet i au sommet j dont tous les sommets intermédiaires se trouvent dans l'ensemble $\{1, 2, \dots, k\}$, et 0 sinon. La fermeture transitive $G^* = (S, A^*)$ est construite en plaçant l'arc (i, j) dans A^* si et seulement si $t_{ij}^{(n)} = 1$. Une définition récursive de $t_{ij}^{(k)}$, analogue à la récurrence (25.5), est

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{si } i \neq j \text{ et } (i, j) \notin A, \\ 1 & \text{si } i = j \text{ ou } (i, j) \in A, \end{cases}$$

et pour $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) . \quad (25.8)$$

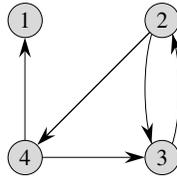
Comme dans l'algorithme de Floyd-Warshall, on calcule les matrices $T^{(k)} = (t_{ij}^{(k)})$ par ordre croissant des valeurs de k .

FERMETURE-TRANSITIVE(G)

```

1   $n \leftarrow |S[G]|$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire pour  $j \leftarrow 1$  à  $n$ 
4          faire si  $i = j$  or  $(i, j) \in A[G]$ 
5              alors  $t_{ij}^{(0)} \leftarrow 1$ 
6              sinon  $t_{ij}^{(0)} \leftarrow 0$ 
7  pour  $k \leftarrow 1$  à  $n$ 
8      faire pour  $i \leftarrow 1$  à  $n$ 
9          faire pour  $j \leftarrow 1$  à  $n$ 
10             faire  $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
11  retourner  $T^{(n)}$ 
```

La figure 25.5 montre les matrices $T^{(k)}$ calculées par la procédure FERMETURE-TRANSITIVE sur un graphe particulier. Comme avec l'algorithme de Floyd-Warshall, le temps d'exécution de la procédure FERMETURE-TRANSITIVE est $\Theta(n^3)$. Toutefois, sur certains ordinateurs, les opérations logiques sur des valeurs à un seul bit s'exécutent plus rapidement que les opérations arithmétiques sur des mots complets. Par ailleurs, comme cet algorithme n'utilise que des valeurs booléennes et non des valeurs entières, ses besoins en mémoire sont diminués par rapport à l'algorithme de Floyd-Warshall d'un facteur correspondant à l'espace occupé par un mot mémoire.



$$\begin{aligned}
 T^{(0)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} & T^{(1)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} & T^{(2)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \\
 T^{(3)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} & T^{(4)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}
 \end{aligned}$$

Figure 25.5 Un graphe orienté et les matrices $T^{(k)}$ calculées par l'algorithme de fermeture transitive.

Exercices

25.2.1 Exécuter l'algorithme de Floyd-Warshall sur le graphe orienté pondéré de la figure 25.2. Donner la matrice $D^{(k)}$ résultant de chaque itération de la boucle extérieure.

25.2.2 Montrer comment calculer la fermeture transitive à l'aide de la technique vue à la section 25.1.

25.2.3 Modifier la procédure FLOYD-WARSHALL pour y inclure le calcul des matrices $\Pi^{(k)}$ exprimées par les équations (25.6) et (25.7). Démontrer rigoureusement que, pour tout $i \in S$, le sous-graphe de liaison $G_{\pi,i}$ est une arborescence de plus courts chemins ayant comme racine i . (conseil : Pour montrer que $G_{\pi,i}$ est sans circuit, commencer par montrer que $\pi_{ij}^{(k)} = l$ implique $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$, selon la définition de $\pi_{ij}^{(k)}$. Ensuite, adapter la démonstration du lemme 24.16.)

25.2.4 Dans la version donnée précédemment, l'algorithme de Floyd-Warshall nécessite de l'espace $\Theta(n^3)$, puisqu'on calcule $d_{ij}^{(k)}$ pour $i, j, k = 1, 2, \dots, n$. Montrer que la procédure suivante, qui se contente de supprimer toutes les puissances, est correcte et qu'il suffit donc d'un espace $\Theta(n^2)$.

FLOYD-WARSHALL'(W)

```

1  n ← lignes[W]
2  D ← W
3  pour k ← 1 à n
4      faire pour i ← 1 à n
5          faire pour j ← 1 à n
6              faire  $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$ 
7  retourner D

```

25.2.5 Supposons qu'on modifie les conditions d'égalité de l'équation (25.7) :

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{si } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{si } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Cette autre définition de la matrice de liaison Π est-elle correcte ?

25.2.6 Comment peut-on utiliser la sortie de l'algorithme de Floyd-Warshall pour détecter la présence d'un circuit de longueur strictement négative ?

25.2.7 Un autre moyen de reconstruire les plus courts chemins dans l'algorithme de Floyd-Warshall consiste à utiliser les valeurs $\phi_{ij}^{(k)}$ pour $i, j, k = 1, 2, \dots, n$, où $\phi_{ij}^{(k)}$ est le dernier sommet intermédiaire d'un plus court chemin de i à j dont tous les sommets intermédiaires sont dans $\{1, 2, \dots, k\}$. Donner une formulation récursive de $\phi_{ij}^{(k)}$, modifier la procédure FLOYD-WARSHALL pour calculer les valeurs $\phi_{ij}^{(k)}$, puis réécrire la procédure IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES pour qu'elle prenne en entrée la matrice $\Phi = (\phi_{ij}^{(n)})$. En quoi la matrice Φ ressemble-t-elle au tableau s du problème de la multiplication des matrices en chaîne vu à la section 15.2 ?

25.2.8 Donner un algorithme en $O(SA)$ permettant de calculer la fermeture transitive d'un graphe orienté $G = (S, A)$.

25.2.9 On suppose que la fermeture transitive d'un graphe orienté sans circuit peut être calculée en $f(|S|, |A|)$, où f est une fonction monotone croissante de $|S|$ et $|A|$. Montrer que le temps de calcul de la fermeture transitive $G^* = (S, A^*)$ d'un graphe orienté quelconque $G = (S, A)$ est $f(|S|, |A|) + O(S + A^*)$.

25.3 ALGORITHME DE JOHNSON POUR LES GRAPHES PEU DENSES

L'algorithme de Johnson trouve les plus courts chemins entre tous les couples de sommets en $O(S^2 \lg S + SA)$; il est donc asymptotiquement plus performant que les élévations au carré répétées de matrices ou que l'algorithme de Floyd-Warshall pour les graphes peu denses. L'algorithme retourne une matrice de longueurs de plus court chemin pour tous les couples de sommets, ou bien indique que le graphe contient un circuit de longueur strictement négative. L'algorithme de Johnson utilise comme sous-programmes à la fois l'algorithme de Dijkstra et l'algorithme de Bellman-Ford, qui sont décrits au chapitre 24.

L'algorithme de Johnson a recours à la technique de *repondération*, qui fonctionne de la manière suivante. Si tous les poids d'arc w d'un graphe $G = (S, A)$ sont positifs ou nuls, on peut trouver les plus courts chemins entre tous les couples de sommets en exécutant l'algorithme de Dijkstra une fois à partir de chaque sommet ; avec une file de priorité min basée sur un tas de Fibonacci, le temps d'exécution de cet algorithme toutes-paires est en $O(S^2 \lg S + SA)$. Si G contient des arcs de poids négatif mais

pas de circuit de longueur strictement négative, on se contente de calculer un nouvel ensemble de poids d'arc positifs qui permettra d'utiliser la même méthode. Le nouvel ensemble de poids d'arc \widehat{w} doit vérifier deux propriétés importantes.

- 1) Pour tout couple de sommets $u, v \in S$, un chemin p est un plus court chemin de u à v utilisant la fonction de pondération w si et seulement si p est aussi un plus court chemin de u à v utilisant la fonction de pondération \widehat{w} .
- 2) Pour tout couple (u, v) , le nouveau poids $\widehat{w}(u, v)$ est positif.

Comme nous allons le voir bientôt, le pré traitement de G servant à déterminer la nouvelle fonction de pondération \widehat{w} peut être réalisé en $O(SA)$.

a) Conservation des plus courts chemins par repondération

Comme le montre le lemme suivant, il est facile de mettre au point une repondération des arcs qui vérifie la première propriété susmentionnée. On utilise δ pour noter les longueurs de plus court chemin déduits de la fonction \widehat{w} , et $\widehat{\delta}$ pour désigner les longueurs de plus court chemin calculés à partir de la fonction de pondération \widehat{w} .

Lemme 25.1 (La repondération ne modifie pas les plus courts chemins) *Étant donné un graphe orienté pondéré $G = (S, A)$ de fonction de pondération $w : A \rightarrow \mathbf{R}$, soit $h : S \rightarrow \mathbf{R}$ une fonction associant à chaque sommet un nombre réel. Pour chaque arc $(u, v) \in A$, on définit*

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v). \quad (25.9)$$

Soit $p = \langle v_0, v_1, \dots, v_k \rangle$ un chemin du sommet v_0 au sommet v_k . Alors, p est un plus court chemin de v_0 à v_k avec la fonction de pondération w si et seulement si c'est un plus court chemin avec la fonction de pondération \widehat{w} . En d'autres termes, $w(p) = \delta(v_0, v_k)$ si et seulement si $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. En outre, G a un circuit de longueur strictement négative utilisant la fonction de pondération w si et seulement si G a un circuit de longueur strictement négative utilisant la fonction de pondération \widehat{w} .

Démonstration : Commençons par montrer que

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k). \quad (25.10)$$

On a

$$\begin{aligned} \widehat{w}(p) &= \sum_{i=1}^k \widehat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ &\quad \text{(car une partie des termes de la somme s'annulent mutuellement)} \\ &= w(p) + h(v_0) - h(v_k). \end{aligned}$$

Par conséquent, tout chemin p entre v_0 et v_k vérifie $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. Si un chemin menant de v_0 à v_k est plus court qu'un autre pour la fonction de pondération w , alors il est également plus court pour \widehat{w} . Donc, $w(p) = \delta(v_0, v_k)$ si et seulement si $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

Enfin, on montre que G contient un circuit de longueur strictement négative pour la fonction w si et seulement si G contient un circuit de longueur strictement négative pour la fonction de pondération \widehat{w} . Considérons un circuit quelconque $c = \langle v_0, v_1, \dots, v_k \rangle$, où $v_0 = v_k$. D'après l'équation (25.10),

$$\begin{aligned}\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c),\end{aligned}$$

et donc c possède une longueur négative pour w si et seulement si il a une longueur négative pour \widehat{w} . \square

b) Obtention de poids positifs par repondération

Notre prochain objectif consiste à faire en sorte que la deuxième propriété soit vraie : on souhaite que $\widehat{w}(u, v)$ soit positive pour tout arc $(u, v) \in A$. Étant donné un graphe orienté pondéré $G = (S, A)$ de fonction de pondération $w : A \rightarrow \mathbf{R}$, on construit un nouveau graphe $G' = (S', A')$, où $S' = S \cup \{s\}$ pour un nouveau sommet $s \notin S$ donné et $A' = A \cup \{(s, v) : v \in S\}$. La fonction de pondération w est étendue de sorte que $w(s, v) = 0$ pour tout $v \in S$. Notez que, comme aucun arc n'entre dans s , aucun plus court chemin de G' , hormis ceux d'origine s , ne contient s . Par ailleurs, G' ne contient aucun circuit de longueur strictement négative si et seulement si G ne contient aucun circuit de longueur strictement négative. La figure 25.6(a) montre le graphe G' qui correspond au graphe G de la figure 25.1.

Supposons à présent que G et G' ne contiennent aucun circuit de longueur strictement négative. On définit $h(v) = \delta(s, v)$ pour tout $v \in S'$. D'après l'inégalité triangulaire 24.10, on a $h(v) \leq h(u) + w(u, v)$ pour tout arc $(u, v) \in A'$. Donc, si l'on définit les nouveaux poids \widehat{w} d'après l'équation (25.9), on a $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, et la seconde propriété est vérifiée. La figure 25.6(b) montre le graphe G' déduit de la figure 25.6(a) après repondération des arcs.

c) Calcul des plus courts chemins pour tout couple de sommets

L'algorithme de Johnson calculant les plus courts chemins pour tout couple de sommets utilise les algorithmes de Bellman-Ford (Section 24.1) et de Dijkstra (Section 24.3) comme sous-programmes. Il suppose que les arcs sont représentés par des listes d'adjacences. L'algorithme retourne la matrice $|S| \times |S|$ habituelle $D = d_{ij}$, où $d_{ij} = \delta(i, j)$, ou bien indique que le graphe contient un circuit de longueur négative. Comme c'est l'habitude pour un algorithme de plus courts chemins toutes-paires, on suppose que les sommets sont numérotés de 1 à $|S|$.)

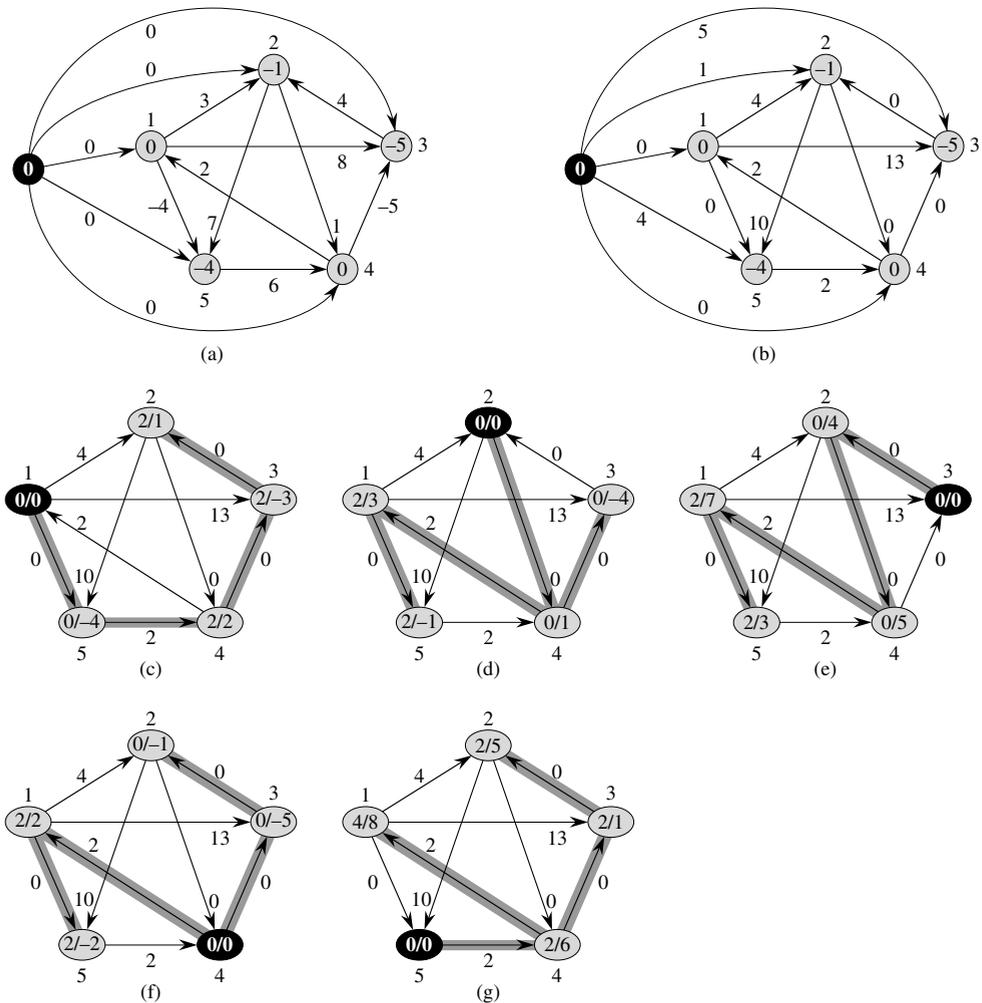


Figure 25.6 L'algorithme de Johnson calculant les plus courts chemins pour tout couple de sommets du graphe de la figure 25.1. (a) Le graphe G' pour la fonction de pondération w initiale. Le nouveau sommet s est noir. A l'intérieur de chaque sommet v est représenté $h(v) = \delta(s, v)$. (b) Chaque arc (u, v) est repondéré avec la fonction de pondération $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. (c)–(g) Le résultat de l'exécution de l'algorithme de Dijkstra sur chaque sommet de G , avec pour fonction de pondération \hat{w} . Dans chaque partie, le sommet origine u est noir et les arcs sur fond gris sont dans l'arborescence de plus courts chemins calculée par l'algorithme. Les valeurs $\hat{\delta}(u, v)$ et $\delta(u, v)$ sont représentées à l'intérieur de chaque sommet v , séparées par un slash. La valeur $d_{uv} = \delta(u, v)$ est égale à $\hat{\delta}(u, v) + h(v) - h(u)$.

JOHNSON(G)

```

1  calculer  $G'$ , où  $S[G'] = S[G] \cup \{s\}$ ,
       $A[G'] = A[G] \cup \{(s, v) : v \in S[G]\}$ , et
       $w(s, v) = 0$  pour tout  $v \in S[G]$ 
2  si BELLMAN-FORD( $G', w, s$ ) = FAUX
3      alors imprimer « le graphe contient un circuit de longueur strictement négative »
4  sinon pour chaque sommet  $v \in S[G']$ 
5      faire affecter à  $h(v)$  la valeur de  $\delta(s, v)$ 
           calculée par l'algorithme de Bellman-Ford
6  pour chaque arc  $(u, v) \in A[G']$ 
7      faire  $\widehat{w}(u, v) \leftarrow w(u, v) + h(u) - h(v)$ 
8  pour chaque sommet  $u \in S[G]$ 
9      faire exécuter DIJKSTRA( $G, \widehat{w}, u$ ) pour calculer  $\widehat{\delta}(u, v)$ 
           pour tout  $v \in S[G]$ 
10     pour chaque sommet  $v \in S[G]$ 
11         faire  $d_{uv} \leftarrow \widehat{\delta}(u, v) + h(v) - h(u)$ 
12     retourner  $D$ 

```

Ce code se contente d'effectuer les actions spécifiées précédemment. La ligne 1 construit G' . La ligne 2 exécute l'algorithme de Bellman-Ford sur G' en utilisant la fonction de pondération w et le sommet origine s . Si G' , et donc G , contient un circuit de longueur strictement négative, la ligne 3 signale le problème. Les lignes 4–11 supposent que G' ne contient aucun circuit de longueur strictement négative. Les lignes 4–5 donnent à $h(v)$ les longueurs de plus court chemin $\delta(s, v)$ calculée par l'algorithme de Bellman-Ford pour tout $v \in S'$. Les lignes 6–7 calculent les nouveaux poids \widehat{w} . Pour tout couple de sommets $u, v \in S$, la boucle **pour** des lignes 8–11 calcule les longueurs de plus court chemin $\widehat{\delta}(u, v)$ en appelant l'algorithme de Dijkstra une fois pour chaque sommet de S . La ligne 11 stocke, dans une composante d_{uv} de la matrice, la longueur de plus court chemin correct $\delta(u, v)$ calculée grâce à l'équation (25.10). Enfin, la ligne 12 retourne la matrice D complète. La figure 25.6 montre l'exécution de l'algorithme de Johnson.

Si la file de priorité min de l'algorithme de Dijkstra est gérée *via* un tas de Fibonacci, le temps d'exécution de l'algorithme de Johnson est $O(S^2 \lg S + SA)$. L'implémentation, plus simple, basée sur un tas min binaire donne un temps d'exécution de $O(SA \lg S)$, qui reste asymptotiquement plus rapide que l'algorithme de Floyd-Warshall si le graphe est peu dense.

Exercices

25.3.1 Utiliser l'algorithme de Johnson pour trouver les plus courts chemins entre tous les couples de sommets du graphe de la figure 25.2. Donner les valeurs de h et \widehat{w} calculées par l'algorithme.

25.3.2 À quoi sert l'ajout du nouveau sommet s à S (ce qui engendre S') ?

25.3.3 Supposez que $w(u, v) \geq 0$ pour tout arc $(u, v) \in A$. Quelle est la relation entre les fonctions de pondération w et \widehat{w} ?

25.3.4 Le professeur Nimbus affirme qu'il existe une façon plus simple, pour repondérer les arcs, que la méthode de Johnson. Soit $w^* = \min_{(u,v) \in A} \{w(u, v)\}$; on se contente de définir $\widehat{w}(u, v) = w(u, v) - w^*$ pour tout arc $(u, v) \in A$. Où est l'erreur dans la méthode de repondération du professeur ?

25.3.5 Supposez que l'on exécute l'algorithme de Johnson sur un graphe orienté G ayant la fonction de pondération w . Montrer que, si G contient un circuit c de poids 0, alors $\widehat{w}(u, v) = 0$ pour tout arc (u, v) de c .

25.3.6 Le professeur Cosinus affirme que ce n'est pas la peine de créer un nouveau sommet origine en ligne 1 de JOHNSON. Il prétend que l'on peut se contenter d'utiliser $G' = G$ et de prendre pour s un quelconque sommet de $S[G]$. Donner un exemple de graphe orienté pondéré G pour lequel la suggestion du professeur donne des réponses incorrectes. Montrer ensuite que, si G est fortement connexe (tout sommet est accessible depuis n'importe quel autre sommet), alors l'idée du professeur donne des résultats corrects.

PROBLÈMES

25.1. Fermeture transitive d'un graphe dynamique

Supposons qu'on veuille préserver la fermeture transitive d'un graphe orienté $G = (S, A)$ malgré l'insertion de nouveaux arcs dans A . En d'autres termes, après chaque insertion d'un arc, on souhaite mettre à jour la fermeture transitive des arcs déjà insérés. On suppose qu'initialement le graphe G ne contient aucun arc, et que la fermeture transitive doit être représentée par une matrice booléenne.

- Montrer comment la fermeture transitive $G^* = (S, A^*)$ d'un graphe $G = (S, A)$ peut être mise à jour en $O(S^2)$ quand un nouvel arc est inséré dans G .
- Donner un exemple de graphe G et d'arc a tel qu'il faille un temps $\Omega(S^2)$ pour mettre à jour la fermeture transitive après l'insertion de a dans G , quel que soit l'algorithme employé.
- Décrire un algorithme efficace de mise à jour de la fermeture transitive, à mesure que des arcs sont insérés dans le graphe. Pour une séquence quelconque de n insertions, le temps d'exécution total de votre algorithme devra être $\sum_{i=1}^n t_i = O(S^3)$, où t_i est le temps nécessaire à la mise à jour de la fermeture transitive quand le i ème arc est inséré. Démontrer que votre algorithme atteint cette borne.

25.2. Plus courts chemins dans un graphe ε -dense

Un graphe $G = (S, A)$ est ε -dense si $|A| = \Theta(S^{1+\varepsilon})$ pour une certaine constante ε de l'intervalle $0 < \varepsilon \leq 1$. En utilisant des tas n -aires (voir problème 6.2) dans les

algorithmes de plus court chemin sur des graphes ε -dense, on peut atteindre les temps d'exécution des algorithmes basés sur des tas de Fibonacci sans avoir à utiliser une structure de données aussi complexe.

- a. Quels sont les temps d'exécution asymptotiques de INSÉRER, EXTRAIRE-MIN et DIMINUER-CLÉ, en fonction de n et du nombre m d'éléments dans un tas n -aire ? Quels sont ces temps d'exécution quand on prend $n = \Theta(m^\alpha)$ pour une certaine constante $0 < \alpha \leq 1$? Comparer ces temps d'exécution avec les coûts amortis de ces opérations pour un tas de Fibonacci.
- b. Montrer comment calculer en $O(A)$ les plus courts chemins à origine unique sur un graphe orienté ε -dense $G = (S, A)$ n'ayant pas d'arc de poids négatif. (*conseil* : Choisir un n qui soit une fonction de ε .)
- c. Montrer comment résoudre en $O(SA)$ le problème des plus courts chemins pour tout couple de sommets sur un graphe orienté ε -dense $G = (S, A)$ n'ayant pas d'arc de poids négatif.
- d. Montrer comment résoudre en $O(SA)$ le problème des plus courts chemins pour tout couple de sommets sur un graphe orienté ε -dense pouvant avoir des arcs de poids négatif, mais pas de circuit de longueur strictement négative.

NOTES

Lawler [196] contient une bonne étude du problème des plus courts chemins toutes-paires, bien qu'il n'analyse pas les solutions pour les graphes peu denses. Il attribue l'algorithme de multiplication matricielle au folklore. L'algorithme de Floyd-Warshall est dû à Floyd[89], qui s'inspira d'un théorème de Warshall [308] expliquant comment calculer la fermeture transitive des matrices booléennes. L'algorithme de Johnson est tiré de [168].

Plusieurs chercheurs ont donné des algorithmes améliorés pour le calcul des plus courts chemins via multiplication de matrices. Fredman [95] montre que le problème des plus courts chemins toutes-paires peut être résolu à l'aide de $O(S^{5/2})$ comparaisons entre sommes de poids d'arc et il obtient un algorithme en $O(S^3(\lg \lg S / \lg S)^{1/3})$, ce qui est un peu mieux que Floyd-Warshall. Une autre ligne de recherche démontre que les algorithmes de multiplication matricielle rapide (voir les notes du chapitre 28) peuvent d'appliquer au problème des plus courts chemins toutes-paires. Soit $O(n^\omega)$ le temps d'exécution de l'algorithme le plus rapide pour la multiplication de matrices $n \times n$ matrices ; pour l'instant, $\omega < 2,376$ [70]. Galil et Margalit [105, 106] et Seidel [270] ont conçu des algorithmes qui résolvent le problème des plus courts chemins toutes-paires, pour les graphes non orientés et non pondérés, en $(S^\omega p(S))$, où $p(n)$ désigne une certaine fonction polylogarithmiquement bornée en n . Dans les graphes denses, ces algorithmes font mieux que le temps $O(SA)$ exigé par l'exécution de $|S|$ recherches en largeur. Plusieurs chercheurs ont étendu ces résultats pour donner des algorithmes de résolution du problème des plus courts chemins toutes-paires dans les graphes non orientés dont les poids des arcs sont des entiers appartenant à l'intervalle $\{1, 2, \dots, W\}$. L'algorithme de ce genre qui est le plus rapide asymptotiquement est dû à Shoshan et à Zwick [278], et il s'exécute en $O(WS^\omega p(SW))$.

Karger, Koller et Phillips [170] et McGeoch [215] indépendamment ont donné une borne temporelle qui dépend de A^* , ensemble des arcs de A qui participent à un certain plus court chemin. Étant donné un graphe ayant des poids d'arc positifs, leurs algorithmes s'exécutent en $O(SA^* + S^2 \lg S)$ et font mieux que l'exécution de l'algorithme de Dijkstra $|S|$ fois quand $|A^*| = o(A)$.

Aho, Hopcroft et Ullman [5] ont défini une structure algébrique dite « semi-anneau fermé », qui sert de cadre général à la résolution des problèmes de chemin dans les graphes orientés. L'algorithme de Floyd-Warshall et l'algorithme de fermeture transitive vu à la section 25.2 sont deux cas particuliers d'algorithme toutes-paires basé sur les semi-anneaux fermés. Maggs et Plotkin [208] ont montré comment trouver des arbres couvrant minimaux à l'aide d'un semi-anneau fermé.

Chapitre 26

Flot maximum

De même qu'il est possible de modéliser une carte routière par un graphe orienté pour trouver le plus court chemin d'un point à un autre, de même on peut interpréter un graphe orienté comme un « réseau de transport » et l'utiliser pour répondre à des questions ayant trait à des flux de produits. Imaginons un produit s'écoulant à travers un système depuis une source, où il est produit, vers un puits, où il est consommé. La source génère le produit avec un certain débit constant et le puits consomme le produit avec le même débit. Le « flot » de produit à un endroit quelconque du système est intuitivement la vitesse à laquelle ce produit se déplace. Les réseaux de transport peuvent servir à modéliser la circulation de liquides à travers des tuyaux, de pièces détachées à travers des chaînes de montage, de courant à travers des réseaux électriques, de données à travers des réseaux de communication, etc.

Chaque arc d'un réseau de flot peut être vu comme un conduit emprunté par le produit. Chaque conduit a une capacité fixe, qui représente le débit maximum que peut atteindre le produit à travers le conduit ; par exemple, 200 litres de liquide par heure dans un tuyau ou 20 ampères de courant à travers un câble. Les sommets sont les jonctions des conduits et, excepté pour la source et le puits, le produit s'écoule d'un sommet à l'autre sans gain ni perte. Autrement dit, le débit à l'entrée d'un sommet doit être égal au débit en sortie. Cette propriété a pour nom « conservation de flot » et équivaut à la loi de Kirchhoff dans le cas du courant électrique.

Le problème du flot maximum est le suivant : on veut connaître la plus grande vitesse à laquelle le produit peut voyager entre la source et le puits, sans violer aucune contrainte de capacité. C'est l'un des problèmes de réseau de transport les plus simples et, comme nous le verrons dans ce chapitre, ce problème peut être résolu

par des algorithmes efficaces. Par ailleurs, les techniques de bases utilisées par ces algorithmes peuvent être adaptées à la résolution d'autres problèmes ayant trait aux réseaux de flot.

Ce chapitre présente deux méthodes générales permettant de résoudre le problème du flot maximum. La section 26.1 formalise les notions de réseaux de transport et de flot, définissant formellement le problème du flot maximum. La section 26.2 décrit la méthode classique de Ford et Fulkerson utilisée pour trouver les flots maxima. Cette méthode trouve une application dans la recherche d'un couplage maximum dans un graphe non orienté biparti, ce qui fait l'objet de la section 26.3. La section 26.4 présente la méthode de préflot, qui est à la base de nombreux algorithmes parmi les plus rapides pour les problèmes de flot. La section 26.5 présente l'algorithme « d'élévation-vers-l'avant », qui est une implémentation particulière de la méthode de préflot s'exécutant en temps $O(S^3)$. Bien que cet algorithme ne soit pas le plus rapide à ce jour, il illustre certaines des techniques utilisées dans les algorithmes les plus rapides asymptotiquement, tout en étant plutôt efficace en pratique.

26.1 RÉSEAUX DE TRANSPORT

Dans cette section, nous allons considérer les réseaux de transport sous l'angle de la théorie des graphes, étudier leurs propriétés et définir précisément le problème du flot maximum. Nous introduisons également quelques notations utiles.

a) Flots et réseaux de transport

Un **réseau de transport** $G = (S, A)$ est un graphe orienté dans lequel chaque arc $(u, v) \in A$ se voit attribuer une **capacité** $c(u, v) \geq 0$. Si $(u, v) \notin A$, on suppose que $c(u, v) = 0$. Dans un réseau de transport, deux sommets ont un statut particulier : la **source** s et le **puits** t . Par commodité, on suppose que chaque sommet se trouve sur un certain chemin reliant la source au puits. Autrement dit, pour tout sommet $v \in S$, il existe un chemin $s \rightsquigarrow v \rightsquigarrow t$. Le graphe est donc connexe et $|A| \geq |S| - 1$. La figure 26.1 montre un exemple de réseau de transport.

Nous pouvons désormais définir les flots plus formellement. Soit $G = (S, A)$ un réseau de transport avec une fonction de capacité c . Soit s la source du réseau et t le puits. Un **flot** de G est une fonction à valeurs réelles $f : S \times S \rightarrow \mathbf{R}$ qui satisfait aux trois propriétés suivantes :

Contrainte de capacité : Pour tout $u, v \in S$, on exige $f(u, v) \leq c(u, v)$.

Symétrie : Pour tout $u, v \in S$, on exige $f(u, v) = -f(v, u)$.

Conservation du flot : Pour tout $u \in S - \{s, t\}$, on exige

$$\sum_{v \in S} f(u, v) = 0.$$

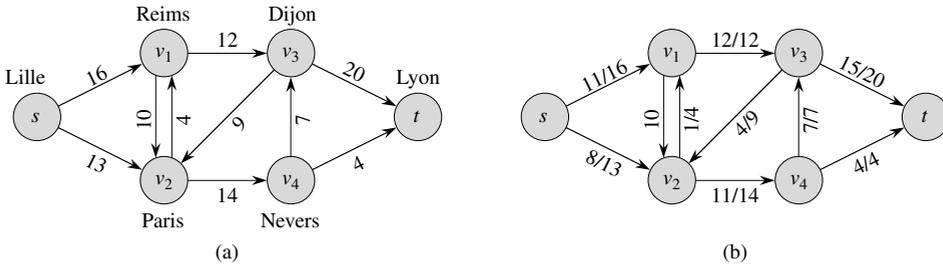


Figure 26.1 (a) Un réseau de transport $G = (S, A)$ pour le problème du transporteur Max & Fils. L'usine de Lille est la source s et l'entrepôt de Lyon est le puits t . Le trajet traverse plusieurs villes intermédiaires, mais on ne peut convoier plus de $c(u, v)$ rouleaux entre les villes u et v . Chaque arc est étiqueté avec sa capacité. (b) Un flot f de G de valeur $|f| = 19$. Seuls les flots positifs du réseau sont montrés. Si $f(u, v) > 0$, l'arc (u, v) est étiqueté par $f(u, v)/c(u, v)$. (La notation slash sert simplement à séparer le flot et la capacité ; elle n'indique pas une division) Si $f(u, v) \leq 0$, l'arc (u, v) est étiqueté uniquement avec sa capacité.

La quantité $f(u, v)$, qui peut être positive, nulle ou négative, est appelée **flux** du sommet u au sommet v . La **valeur** d'un flot f est définie par

$$|f| = \sum_{v \in S} f(s, v), \tag{26.1}$$

autrement dit, le flot total partant de la source. (Ici, la notation $|\cdot|$ indique la valeur de flot, pas une valeur absolue ni un cardinal d'ensemble). Dans le **problème du flot maximum**, on part d'un réseau de flot G de source s et de puits t et on souhaite trouver un flot de valeur maximale.

Avant de voir un exemple de problème de flot, arrêtons-nous brièvement sur les trois propriétés de flot. La contrainte de capacité dit simplement que le flux d'un sommet vers un autre ne doit pas excéder la capacité donnée. La propriété de symétrie est une commodité notationnelle qui dit que le flux d'un sommet u vers un sommet v est égal à l'opposé du flux allant en sens inverse. La propriété de conservation du flot dit que le flot total sortant d'un sommet autre que la source ou le puits a pour valeur 0. En s'aidant de la symétrie, on peut réécrire la propriété de conservation du flot sous la forme

$$\sum_{u \in S} f(u, v) = 0$$

pour tout $v \in S - \{s, t\}$. Autrement dit, le flot total entrant dans un sommet vaut 0.

Si ni (u, v) ni (v, u) n'est dans A , il ne peut pas y avoir de flot entre u et v et $f(u, v) = f(v, u) = 0$. (L'exercice 26.1.1 vous demandera de prouver cette propriété formellement.)

Notre dernière observation sur les propriétés de flot concerne les flots à valeur positive. Le **flux positif total** entrant dans un sommet v est défini par

$$\sum_{\substack{u \in S \\ f(u,v) > 0}} f(u,v). \quad (26.2)$$

Le flux positif total sortant d'un sommet est défini symétriquement. Le **flux net total** en un sommet est égal au flux positif total sortant du sommet, moins le flux positif total entrant dans le sommet. Une interprétation de la propriété de conservation du flux est que le flux positif total entrant dans un sommet autre que la source ou le puits doit être égal au flux positif total sortant du sommet. Cette propriété, selon laquelle le flux net total en un sommet doit valoir 0, porte souvent le nom informel de « flux entrant égale flux sortant ».

b) Exemple de flot

Un réseau de flot peut modéliser le problème de convoyage illustré à la figure 26.1. L'entreprise Max & Fils possède une usine (source s) à Lille qui fabrique du tissu et un entrepôt (puits t) à Lyon, qui permet de le stocker. Max & Fils fait appel à une autre entreprise pour convoier par camion le tissu entre l'usine et l'entrepôt. Comme les camions ne peuvent pas emprunter n'importe quelle route (arc) et ont des tonnages limités, Max & Fils peut convoier au plus $c(u,v)$ rouleaux par jour entre chaque paire de villes u et v de la figure 26.1(a). Max & Fils ne peut décider ni des routes empruntées ni des limites de tonnage, et ne peut donc pas non plus modifier le réseau de transport montré à la figure 26.1(a). Son but est de déterminer le plus grand nombre p de rouleaux qui peuvent être convoyés par jour et ensuite produire cette quantité, puisqu'il est inutile de produire plus de tissu qu'il n'en sera convoyé vers l'entrepôt. Max & Fils n'est pas concerné par le temps requis pour acheminer un rouleau donné entre la fabrique et l'entrepôt ; il s'occupe uniquement de faire sortir p rouleaux par jour de l'usine et de faire arriver p rouleaux par jour à l'entrepôt.

Superficiellement, il semble approprié de modéliser le « flux » de livraisons par un flot de transport ; en effet, le nombre de rouleaux expédiés chaque jour d'une ville à l'autre est soumis à une contrainte de capacité. En outre, il faut respecter la conservation de flot car, dans un état stable, le débit de produits entrants dans une ville intermédiaire doit être égal au débit de produits qui en partent. Sinon, les produits s'accumuleraient dans les villes intermédiaires.

Il y a une différence subtile entre les livraisons et les flots, toutefois. Max et Fils peut expédier du tissu de Reims à Paris, mais peut aussi faire des expéditions en sens inverse. Supposez qu'ils expédient 8 rouleaux par jour de Reims (v_1 sur la figure 26.1) à Paris (v_2), et 3 rouleaux par jour de Paris à Reims. Il peut sembler naturel de représenter ces expéditions directement par des flux, mais c'est impossible. La contrainte de symétrie exige que $f(v_1, v_2) = -f(v_2, v_1)$, ce qui n'est visiblement pas le cas si l'on a $f(v_1, v_2) = 8$ et $f(v_2, v_1) = 3$.

Max et Fils pourrait comprendre qu'il est inutile d'expédier 8 rouleaux par jour de Reims à Paris et 3 rouleaux par jour de Paris à Reims, alors qu'il suffit d'expédier 5 rouleaux de Reims à Paris et 0 rouleaux de Paris à Reims (cela consommerait en outre moins de ressources, en principe). Représentons ce dernier scénario par un flux : on a $f(v_1, v_2) = 5$ et $f(v_2, v_1) = -5$. En fait, 3 des 8 rouleaux quotidiens de v_1 à v_2 sont **annulés** par 3 rouleaux quotidiens de v_2 à v_1 .

En général, l'annulation permet de représenter les expéditions entre deux villes par un flux qui est positif le long d'un au plus des deux arcs reliant les sommets concernés. En clair : toute situation dans laquelle il y a des expéditions dans les deux sens peut être transformée, moyennant annulation, en une situation équivalente dans laquelle les expéditions se font dans un seul sens (le sens du flux positif).

Étant donné un flot f engendré, par exemple, par des expéditions physiques, on ne peut pas reconstruire les expéditions exactes. Si l'on sait que $f(u, v) = 5$, cela peut venir de ce que 5 unités ont été expédiées de u à v , ou de ce que 8 unités ont été expédiées de u à v et 3 de v à u . En général, on ne se soucie pas des détails concrets sous-jacents à la circulation des produits ; pour chaque paire de sommets, on s'intéresse uniquement au montant net qui circule entre ces deux sommets. Si, en revanche, l'on se soucie des détails concrets sous-jacents au transport, alors il faut employer un modèle différent, un qui conserve des informations sur les expéditions dans les deux sens.

L'annulation sera toujours implicite dans les algorithmes de ce chapitre. Supposez que l'arc (u, v) ait une valeur de flux $f(u, v)$. Dans le cours d'un algorithme, on peut accroître le flux sur l'arc (v, u) d'une certaine quantité d . Mathématiquement, cette opération doit décroître $f(u, v)$ de d et, sur le plan conceptuel, on peut penser que ces d unités annulent d unités de flux qui sont déjà sur l'arc (u, v) .

c) Réseaux à sources et puits multiples

Un problème de flot maximum peut comporter plusieurs sources et puits. L'entreprise Max & Fils, par exemple, pourrait en réalité posséder un ensemble $\{s_1, s_2, \dots, s_m\}$ de m usines et un ensemble $\{t_1, t_2, \dots, t_n\}$ de n entrepôts, comme le montre la figure 26.2(a). Heureusement, ce problème n'est pas plus difficile que celui à source et puits uniques.

Le problème du flot maximum à sources et puits multiples peut être réduit à un problème de transport maximum ordinaire. La figure 26.2(b) montre comment le réseau (a) peut être converti en un réseau de transport ordinaire, avec une source et un puits uniques. On ajoute une **supersource** s et un arc (s, s_i) de capacité $c(s, s_i) = \infty$ pour chaque $i = 1, 2, \dots, m$. On crée également un nouveau **superpuits** t et on ajoute un arc (t_i, t) de capacité $c(t_i, t) = \infty$ pour chaque $i = 1, 2, \dots, n$. Intuitivement, tout flot du réseau (a) correspond à un flot du réseau (b) et *vice versa*. La source unique s fournit autant de flot que désiré aux multiples sources s_i et le puits unique t consomme de la même façon autant de flot que désiré par les puits multiples t_i . L'exercice 26.1.3 vous demande de démontrer formellement que les deux problèmes sont équivalents.

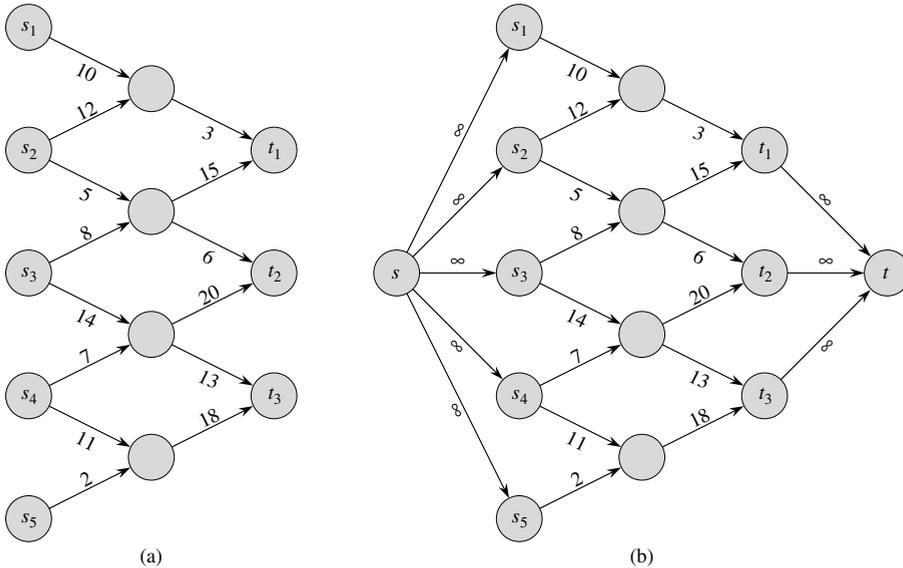


Figure 26.2 Conversion d'un problème à sources et puits multiples en un problème à source et puits uniques. (a) Un réseau transport comportant cinq sources $S = \{s_1, s_2, s_3, s_4, s_5\}$ et trois puits $T = \{t_1, t_2, t_3\}$. (b) Un réseau de transport équivalent avec une seule source et un seul puits. On ajoute une supersource s' et un arc de capacité infinie partant de s' et reliant chacune des multiples sources. On ajoute également un superpuits t' et un arc de capacité infinie partant de chacun des puits initiaux et reliant t' .

d) Travail avec les réseaux

Nous allons travailler avec plusieurs fonctions (comme f) qui prennent comme argument deux sommets d'un réseau de transport. Dans ce chapitre, nous allons utiliser une **notation de sommation implicite** dans laquelle l'un ou l'autre des arguments, ou les deux, peut être un *ensemble* de sommets, avec pour interprétation que la valeur indiquée représente la somme de toutes les façons possibles de remplacer les arguments par leurs membres. Par exemple, si X et Y sont des ensembles de sommets, alors

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y).$$

Ainsi, la conservation de flot peut être exprimée comme la condition $f(u, S) = 0$ pour tout $u \in S - \{s, t\}$. Par commodité, les accolades seront généralement omises dans cette notation, là où elles devraient être utilisées. Par exemple, dans l'équation $f(s, S - s) = f(s, S)$, le terme $S - s$ représente l'ensemble $S - \{s\}$.

La notation ensembliste implicite simplifie souvent les équations mettant en jeu des flots. Le lemme suivant, dont la démonstration est laissée en exercice (exercice 26.1.4), reprend quelques-unes des identités les plus courantes mettant en jeu les flots et la notation ensembliste implicite.

Lemme 26.1 Soit $G = (S, A)$ un réseau de transport et soit f un flot de G . On a alors les égalités suivantes :

- 1) Pour tout $X \subseteq S$, on a $f(X, X) = 0$.
- 2) Pour tout $X, Y \subseteq S$, on a $f(X, Y) = -f(Y, X)$.
- 3) Pour tout $X, Y, Z \subseteq S$ avec $X \cap Y = \emptyset$, on a les sommes $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ et $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

Comme exemple de travail avec la notation implicite, on peut démontrer que la valeur d'un flot est égale au flot total arrivant dans le puits ; c'est-à-dire,

$$|f| = f(S, t) . \quad (26.3)$$

Cela paraît vrai intuitivement. En effet, d'après la conservation du flot, tous les sommets autres que la source et le puits ont un flux net total égal à zéro. La source a, par définition, un flux net total supérieur à 0 : il y a plus de flux positif qui part de la source qu'il n'y a de flux qui arrive à la source. Pour le puits, c'est exactement l'inverse : le flux net total est négatif, car il y a plus de flux positif qui arrive au puits qu'il n'y a de flux positif qui en part. Notre démonstration formelle se présente comme suit :

$$\begin{aligned} |f| &= f(s, S) && \text{(par définition)} \\ &= f(S, S) - f(S - s, S) && \text{(d'après lemme 26.1, partie (3))} \\ &= -f(S - s, S) && \text{(d'après lemme 26.1, partie (1))} \\ &= f(S, S - s) && \text{(d'après lemme 26.1, partie (2))} \\ &= f(S, t) + f(S, S - s - t) && \text{(d'après lemme 26.1, partie (3))} \\ &= f(S, t) && \text{(d'après conservation du flot) .} \end{aligned}$$

Plus loin dans ce chapitre, nous généraliserons ce résultat (lemme 26.5).

Exercices

26.1.1 En utilisant la définition d'un flot, prouver que si $(u, v) \notin A$ et $(v, u) \notin A$, alors $f(u, v) = f(v, u) = 0$.

26.1.2 Prouver que, pour tout sommet v autre que la source ou le puits, le flux positif total arrivant à v doit être égal au flux positif total partant de v .

26.1.3 Etendre les propriétés et les définitions de flot au problème à sources et puits multiples. Montrer que tout flot d'un réseau à sources et puits multiples correspond à un flot de valeur identique du réseau à source et puits uniques obtenu en ajoutant une supersource et un superpuits, et vice versa.

26.1.4 Démontrer le lemme 26.1.

26.1.5 Pour le réseau de transport $G = (S, A)$ et le flot f de la figure 26.1(b), trouver deux sous-ensembles $X, Y \subseteq S$ pour lesquels $f(X, Y) = -f(V - X, Y)$. Puis, trouver deux sous-ensembles $X, Y \subseteq S$ pour lesquels $f(X, Y) \neq -f(S - X, Y)$.

26.1.6 Étant donné un réseau de transport $G = (S, A)$, soient f_1 et f_2 deux fonctions de $S \times S$ vers \mathbf{R} . La *somme des flots* $f_1 + f_2$ est la fonction de $S \times S$ vers \mathbf{R} définie par

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \quad (26.4)$$

pour tout $u, v \in S$. Si f_1 et f_2 sont des flots de G , laquelle des trois propriétés de flot la somme de flots $f_1 + f_2$ doit-elle satisfaire et laquelle pourrait-elle violer ?

26.1.7 Soit f un flot dans un réseau et soit α un nombre réel. Le *produit scalaire-flot* αf est une fonction de $S \times S$ vers \mathbf{R} définie par

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Démontrer que les flots d'un réseau forment un ensemble convexe. C'est-à-dire : montrer que, si f_1 et f_2 sont deux flots, alors $\alpha f_1 + (1 - \alpha)f_2$ est aussi un flot pour tout α dans l'intervalle $0 \leq \alpha \leq 1$.

26.1.8 Énoncer le problème du flot maximum sous la forme d'un problème de programmation linéaire.

26.1.9 Le professeur Adam a deux enfants qui, hélas, se détestent. Ils se haïssent à tel point que chacun refuse d'aller à l'école en empruntant un trottoir sur lequel l'autre a marché le jour même. Les enfants ne font pas de difficultés si leurs chemins se croisent à un coin de rue. Heureusement, la maison du professeur et l'école sont situées à des coins de rue ; à part cela, le professeur ne sait pas s'il pourra envoyer ses deux enfants à la même école. Le professeur a un plan de la ville. Montrer comment formuler en tant que problème de flot maximum le problème consistant à savoir si les deux enfants pourront aller à la même école.

26.2 LA MÉTHODE DE FORD-FULKERSON

Cette section présente la méthode de Ford-Fulkerson, qui permet de résoudre le problème du flot maximum. Nous préférons l'appeler « méthode » plutôt que « algorithme », car elle comporte plusieurs implémentations de temps d'exécution différents. La méthode de Ford-Fulkerson dépend de trois concepts majeurs, qui transcendent la méthode et sont applicables à de nombreux algorithmes et problèmes de flot ; ces concepts sont les réseaux résiduels, les chemins améliorant et les coupes. Ces idées sont essentielles pour le théorème important du flot maximum & coupe minimum (théorème 26.7), qui caractérise la valeur d'un flot maximum en terme de coupes du réseau de transport. Cette section se terminera avec une présentation d'une implémentation particulière de la méthode de Ford-Fulkerson et l'analyse de son temps d'exécution.

La méthode de Ford-Fulkerson est itérative. On commence avec $f(u, v) = 0$ pour tout $u, v \in S$, ce qui donne un flot initial de valeur 0. A chaque itération, on augmente la valeur de flot en trouvant un « chemin améliorant », qu'on peut voir simplement comme un chemin reliant la source s au puits t le long duquel on peut augmenter la quantité de flot. On réitère ce processus jusqu'à ce que l'on ne trouve plus de chemin améliorant. Le théorème du flot maximum & coupe minimum montre que ce processus finit par engendrer un flot maximum.

MÉTHODE-FORD-FULKERSON(G, s, t)

- 1 initialiser flot f à 0
- 2 **tant que** il existe un chemin améliorant p
- 3 **faire** augmenter le flot f le long de p
- 4 **retourner** f

a) Réseaux résiduels

Intuitivement, étant donné un réseau de transport et un flot, le réseau résiduel est constitué des arcs qui peuvent supporter un flot plus important. Plus formellement, supposons qu'on ait un réseau de flux $G = (S, A)$ de source s et de puits t . Soit f un flot de G et considérons un couple de sommets $u, v \in S$. La quantité de flux *supplémentaire* qu'il est possible d'ajouter entre u et v sans dépasser la capacité $c(u, v)$ est la **capacité résiduelle** de (u, v) , donnée par

$$c_f(u, v) = c(u, v) - f(u, v). \quad (26.5)$$

Par exemple, si $c(u, v) = 16$ et $f(u, v) = 11$, on peut convoier $c_f(u, v) = 5$ unités de flux supplémentaires sans excéder la contrainte de capacité sur l'arc (u, v) . Quand le flux net $f(u, v)$ est négatif, la capacité résiduelle $c_f(u, v)$ est supérieure à la capacité $c(u, v)$. Par exemple, si $c(u, v) = 16$ et $f(u, v) = -4$, la capacité résiduelle $c_f(u, v)$ vaut 20. On peut interpréter cela de la manière suivante : il existe un flux de 4 unités de v vers u , qui peut être annulé en envoyant un flux de 4 unités de u vers v . On peut ensuite envoyer 16 unités supplémentaires de u vers v sans violer la contrainte de capacité sur l'arc (u, v) . On a donc pu ajouter 20 unités de flux, en commençant avec un flux $f(u, v) = -4$, avant d'atteindre la contrainte de capacité.

Étant donné un réseau de transport $G = (S, A)$ et un flot f , le **réseau résiduel** de G induit par f est $G_f = (S, A_f)$, où

$$A_f = \{(u, v) \in S \times S : c_f(u, v) > 0\} .$$

Autrement dit, comme promis, chaque arc du réseau résiduel, ou **arc résiduel**, peut admettre un flux strictement positif. La figure 26.3(a) reprend le réseau de transport G et le flot f de la figure 26.1(b) et la figure 26.3(b) montre le réseau résiduel correspondant G_f .

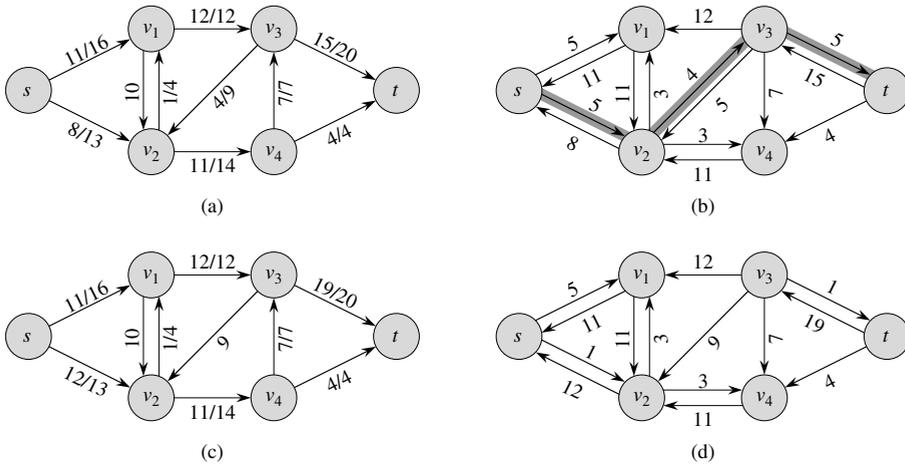


Figure 26.3 (a) Le réseau de transport G et le flot f de la figure 26.1(b). (b) Le réseau résiduel G_f avec le chemin améliorant p en gris ; sa capacité résiduelle est $c_f(p) = c_f(v_2, v_3) = 4$. (c) Le flot de G après ajout le long de p de sa capacité résiduelle 4. (d) Le réseau résiduel induit par le flot de (c).

Les arcs de A_f sont soit des arcs de A , soit leurs inverses. Si $f(u, v) < c(u, v)$ pour un arc $(u, v) \in A$, alors $c_f(u, v) = c(u, v) - f(u, v) > 0$ et $(u, v) \in A_f$. Si $f(u, v) > 0$ pour un arc $(u, v) \in A$, alors $f(v, u) < 0$. Dans ce cas, $c_f(v, u) = c(v, u) - f(v, u) > 0$, et donc $(v, u) \in A_f$. Si ni (u, v) ni (v, u) n'apparaît dans le réseau original, alors $c(u, v) = c(v, u) = 0$, $f(u, v) = f(v, u) = 0$ (d'après l'exercice 26.1.1) et $c_f(u, v) = c_f(v, u) = 0$. On en conclut qu'un arc (u, v) ne peut apparaître dans un réseau résiduel que si au moins l'un des arcs (u, v) et (v, u) figure dans le réseau original ; on a donc

$$|A_f| \leq 2 |A| .$$

Observez que le réseau résiduel G_f est lui-même un réseau de transport dont les capacités sont données par c_f . Le lemme suivant montre la relation entre un flot d'un réseau résiduel et un flot du réseau originel.

Lemme 26.2 Soit $G = (S, A)$ un réseau de transport de source s et de puits t et soit f un flot de G . Soit G_f le réseau résiduel de G induit par f et soit f' un flot de G_f . Alors, la somme $f + f'$ définie par l'équation (26.4) est un flot de G de valeur $|f + f'| = |f| + |f'|$.

Démonstration : Il faut vérifier que la propriété de symétrie, les contraintes de capacité et la conservation de flot sont respectées. Pour la symétrie, notez que pour tout couple $u, v \in S$, on a

$$\begin{aligned} (f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -(f(v, u) + f'(v, u)) \\ &= -(f + f')(v, u) . \end{aligned}$$

Pour les contraintes de capacité, on remarque que $f'(u, v) \leq c_f(u, v)$ pour tout couple $u, v \in S$. Donc, d'après l'équation (26.5),

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\leq f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v) .\end{aligned}$$

Pour la conservation de flot, notez que pour tout $u \in S - \{s, t\}$, on a

$$\begin{aligned}\sum_{v \in S} (f + f')(u, v) &= \sum_{v \in S} (f(u, v) + f'(u, v)) \\ &= \sum_{v \in S} f(u, v) + \sum_{v \in S} f'(u, v) \\ &= 0 + 0 \\ &= 0 .\end{aligned}$$

Enfin, on a

$$\begin{aligned}|f + f'| &= \sum_{v \in S} (f + f')(s, v) \\ &= \sum_{v \in S} (f(s, v) + f'(s, v)) \\ &= \sum_{v \in S} f(s, v) + \sum_{v \in S} f'(s, v) \\ &= |f| + |f'| .\end{aligned}$$

□

b) Chemins améliorants

Étant donné un réseau de transport $G = (S, A)$ et un flot f , un **chemin améliorant** p est un chemin élémentaire de s vers t dans le réseau résiduel G_f . D'après la définition du réseau résiduel, chaque arc (u, v) d'un chemin améliorant admet un flot positif supplémentaire de u vers v tout en restant soumis à la contrainte de capacité sur cet arc.

Le chemin en gris sur la figure 26.3(b) est un chemin améliorant. En considérant le réseau résiduel G_f de la figure comme un réseau de transport, on peut faire passer jusqu'à 4 unités de flux supplémentaire à travers chaque arc de ce chemin, sans violer de contrainte de capacité, puisque la plus petite capacité résiduelle sur ce chemin est $c_f(v_2, v_3) = 4$. La plus grande quantité de flux transportable à travers les arcs d'un chemin améliorant p s'appelle la **capacité résiduelle** de p , et est définie par

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ appartient à } p\} .$$

Le lemme suivant, dont la démonstration est laissée en exercice (voir exercice 26.2.7), précise la définition précédente.

Lemme 26.3 Soit $G = (S, A)$ un réseau de transport, soit f un flot de G et soit p un chemin améliorant de G_f . On définit une fonction $f_p : S \times S \rightarrow \mathbf{R}$ par

$$f_p(u, v) = \begin{cases} c_f(p) & \text{si } (u, v) \text{ appartient à } p, \\ -c_f(p) & \text{si } (v, u) \text{ appartient à } p, \\ 0 & \text{sinon.} \end{cases} \quad (26.6)$$

Alors, f_p est un flot de G_f de valeur $|f_p| = c_f(p) > 0$.

Le corollaire suivant montre que si l'on ajoute f_p à f , on obtient un autre flot de G dont la valeur est plus proche du maximum. La figure 26.3(c) montre les conséquences de l'ajout du f_p de la figure 26.3(b) au f de la figure 26.3(a).

Corollaire 26.4 Soit $G = (S, A)$ un réseau de transport, soit f un flot de G et soit p un chemin améliorant de G_f . Soit f_p une fonction définie comme dans l'équation (26.6). On définit une fonction $f' : S \times S \rightarrow \mathbf{R}$ par $f' = f + f_p$. Alors, f' est un flot de G de valeur $|f'| = |f| + |f_p| > |f|$.

Démonstration : Immédiate d'après les lemmes 26.2 et 26.3. □

c) Coupe dans un réseau de transport

La méthode de Ford-Fulkerson augmente progressivement le flot dans les chemins améliorant, jusqu'à atteindre un flot maximum. Le théorème « flot maximum & coupe minimum », que nous allons bientôt démontrer, nous apprend qu'un flot est maximum si et seulement si son réseau résiduel ne contient aucun chemin améliorant. Pour démontrer ce théorème, il faut cependant commencer par définir la notion de coupe dans un réseau de transport.

Une **coupe** (E, T) d'un réseau de transport $G = (S, A)$ est une partition de S dans E et $T = S - E$ telle que $s \in E$ et $t \in T$. (Cette définition ressemble à celle que nous avons utilisée pour la « coupe » dans un arbre couvrant minimum au chapitre 23, hormis qu'ici, nous coupons un graphe orienté plutôt qu'un graphe non orienté et que nous insistons sur le fait que $s \in E$ et $t \in T$.) Si f est un flot, alors le **flot net** à travers la coupure (E, T) est défini par $f(E, T)$. La **capacité** de la coupe (E, T) est $c(E, T)$. Une **coupe minimum** d'un réseau est une coupe dont la capacité est minimale rapportée à toutes les coupes du réseau.

La figure 26.4 montre la coupe $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ dans le réseau de transport de la figure 26.1(b). Le flot net à travers cette coupe est

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19, \end{aligned}$$

et sa capacité est

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26. \end{aligned}$$

Observez que le flot à travers une coupe peut inclure des flux négatifs entre sommets, mais que la capacité d'une coupe est entièrement constituée de valeurs positives ou nulles. En d'autres termes, le flot net à travers une coupe (E, T) se compose de flux positifs dans les deux sens ; le flux positif de E vers T est ajouté, alors que le flux positif de T vers E est soustrait. En revanche, la capacité d'une coupe (E, T) se calcule uniquement à partir des arcs allant de E à T . Les arcs reliant T à E n'entrent pas en ligne de compte pour le calcul de $c(S, T)$.

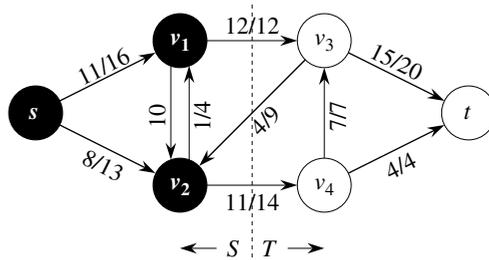


Figure 26.4 Une coupe (E, T) dans le réseau de transport de la figure 26.1(b), où $E = \{s, v_1, v_2\}$ et $T = \{v_3, v_4, t\}$. Les sommets de E sont en noir et ceux de T sont en blanc. Le flot net à travers (E, T) est $f(E, T) = 19$, et la capacité est $c(S, T) = 26$.

Le lemme suivant montre que le flot net traversant une coupe est toujours le même, et qu'il est égal à la valeur du flot.

Lemme 26.5 Soit f un flot dans un réseau de transport G de source s et de puits t et soit (E, T) une coupe de G . Alors, le flot net à travers (E, T) est $f(E, T) = |f|$.

Démonstration : En notant que $f(E - s, V) = 0$ en vertu de la conservation du flot, on a

$$\begin{aligned}
 f(E, T) &= f(E, S) - f(E, E) && \text{(d'après lemme 26.1, partie (3))} \\
 &= f(E, S) && \text{(d'après lemme 26.1, partie (1))} \\
 &= f(s, S) + f(E - s, S) && \text{(d'après lemme 26.1, partie (3))} \\
 &= f(s, S) && \text{(car } f(E - s, S) = 0) \\
 &= |f|.
 \end{aligned}$$

□

Un corollaire immédiat au lemme 26.5 est le résultat démontré précédemment (équation (26.3)), qui établit que la valeur d'un flot est le flot total arrivant dans le puits.

Un autre corollaire au lemme 26.5 montre comment les capacités des coupes peuvent servir à borner la valeur d'un flot.

Corollaire 26.6 La valeur d'un flot f dans un réseau de transport G est majorée par la capacité d'une coupe quelconque de G .

Démonstration : Soit (E, T) une coupe quelconque de G et soit f un flot quelconque. D'après le lemme 26.5 et les contraintes de capacité,

$$\begin{aligned} |f| &= f(E, T) \\ &= \sum_{u \in E} \sum_{v \in T} f(u, v) \\ &\leq \sum_{u \in E} \sum_{v \in T} c(u, v) \\ &= c(E, T). \end{aligned}$$

□

Une conséquence immédiate du corollaire 26.6 est que le flot maximum d'un réseau est majoré par la capacité d'une coupe minimum du réseau. L'important théorème flot maximum & coupe minimum, que nous allons à présent établir et démontrer, dit que la valeur d'un flot maximum est en fait égale à la capacité d'une coupe minimum.

Théorème 26.7 (flot maximum & coupe minimum)

Si f est un flot dans un réseau de transport $G = (S, A)$ de source s et de puits t , alors les conditions suivantes sont équivalentes :

- 1) f est un flot maximum dans G .
- 2) Le réseau résiduel G_f ne contient aucun chemin améliorant.
- 3) $|f| = c(E, T)$ pour une certaine coupe (E, T) de G .

Démonstration : (1) \Rightarrow (2) : supposons, en raisonnant par l'absurde, que f soit un flot maximum de G mais que G_f contienne un chemin améliorant p . Alors, d'après le corollaire 26.4, la somme $f + f_p$, où f_p est donné par l'équation (26.6), est un flot de G dont la valeur est strictement supérieure à $|f|$, ce qui contredit l'hypothèse selon laquelle f est un flot maximum.

(2) \Rightarrow (3) : supposons que G_f n'ait pas de chemin améliorant, c'est-à-dire que G_f ne contienne aucun chemin de s vers t . On définit

$$E = \{v \in S : \text{il existe un chemin de } s \text{ vers } v \text{ dans } G_f\}$$

et $T = S - E$. La partition (E, T) est une coupe : on a $s \in E$ de façon triviale et $t \notin E$ car il n'existe aucun chemin de s vers t dans G_f . Pour chaque couple de sommets u et v tel que $u \in E$ et $v \in T$, on a $f(u, v) = c(u, v)$, car sinon on aurait $(u, v) \in A_f$ et v appartiendrait à E . D'après le lemme 26.5, on a donc $|f| = f(S, T) = c(S, T)$.

(3) \Rightarrow (1) : D'après le corollaire 26.6, $|f| \leq c(E, T)$ pour toutes les coupes (E, T) . La condition $|f| = c(E, T)$ implique donc que f est un flot maximum. □

d) Algorithme de base de Ford-Fulkerson

A chaque itération de la méthode de Ford-Fulkerson, on trouve un *certain* chemin améliorant p et on augmente le flux f sur chaque arc de p d'un montant égal à la capacité résiduelle $c_f(p)$. L'implémentation suivante de cette méthode calcule le flot maximum dans un graphe $G = (S, A)$, en actualisant le flux $f[u, v]$ entre chaque couple

u, v de sommets qui sont reliés par un arc ⁽¹⁾. Si u et v ne sont reliés par aucun arc dans aucune direction, on suppose implicitement que $f[u, v] = 0$. Les capacités $c(u, v)$ sont censées être données avec le graphe, et $c(u, v) = 0$ si $(u, v) \notin A$. La capacité résiduelle $c_f(u, v)$ est calculée selon la formule (26.5). L'expression $c_f(p)$ dans le code n'est en fait qu'une variable temporaire qui mémorise la capacité résiduelle du chemin p .

```

FORD-FULKERSON( $G, s, t$ )
1  pour chaque arc  $(u, v) \in A[G]$ 
2      faire  $f[u, v] \leftarrow 0$ 
3           $f[v, u] \leftarrow 0$ 
4  tant que il existe un chemin  $p$  de  $s$  à  $t$  dans le réseau résiduel  $G_f$ 
5      faire  $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
6          pour chaque arc  $(u, v)$  de  $p$ 
7              faire  $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
8                   $f[v, u] \leftarrow -f[u, v]$ 

```

L'algorithme FORD-FULKERSON se contente de développer le pseudo code FORD-FULKERSON-MÉTHODE donné en amont. La figure 26.5 montre le résultat de chaque itération dans un exemple d'exécution. Les lignes 1–3 initialisent le flot f à 0. La boucle **tant que** des lignes 4–8 trouve de manière répétitive un chemin améliorant p de G_f et augmente le flot f le long de p d'une quantité égale à la capacité résiduelle $c_f(p)$. Quand il n'existe plus aucun chemin augmentant, le flot f est maximum.

e) Analyse de Ford-Fulkerson

Le temps d'exécution de FORD-FULKERSON dépend de la façon dont on détermine le chemin améliorant p à la ligne 4. En cas de choix malheureux, l'algorithme peut même ne jamais se terminer : la valeur du flot augmente par paliers, mais elle n'est pas certaine de converger vers la valeur de flot maximum. ⁽²⁾ En revanche, si le chemin améliorant est choisi à l'aide d'une recherche en largeur (voir section 22.2), l'algorithme s'exécute dans un temps polynomial. Avant de le démontrer, nous allons chercher une borne simple pour le cas où le chemin améliorant est choisi arbitrairement et où toutes les capacités sont entières.

Le plus souvent en pratique, le problème du flot maximum est posé avec des valeurs entières pour les capacités. Lorsque les capacités sont des nombres rationnels, une mise à l'échelle idoine permet de les rendre entières. Avec cette hypothèse, une implémentation directe de FORD-FULKERSON s'exécute en temps $O(A |f^*|)$, où f^* est le flot maximum trouvé par l'algorithme. L'analyse est la suivante. Les lignes 1–3 prennent un temps $\Theta(A)$. La boucle **tant que** des lignes 4–8 est exécutée au plus $|f^*|$ fois, puis la valeur du flot augmente d'au moins une unité à chaque itération.

(1) Nous utiliserons des crochets quand un identificateur, tel f , est traité comme un champ variable, et des parenthèses quand il est traité comme une fonction.

(2) La méthode de Ford-Fulkerson n'arrive pas à se terminer que si les capacités d'arc sont des nombres irrationnels. En pratique, toutefois, un ordinateur a une précision finie et ne peut pas stocker de nombres irrationnels.

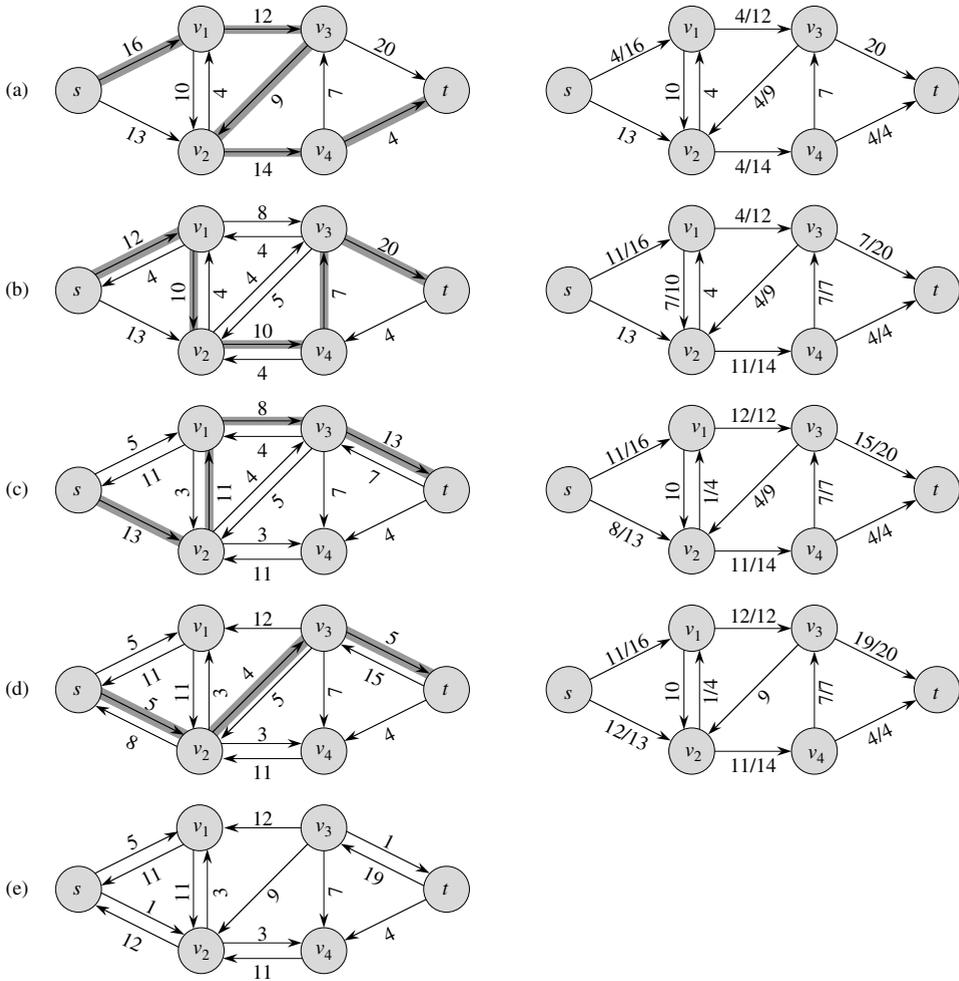


Figure 26.5 L'exécution de l'algorithme de base de Ford-Fulkerson. (a)–(d) Itérations successives de la boucle **tant que**. Le côté gauche de chaque partie montre le réseau résiduel G_f de la ligne 4 avec un chemin améliorant p en gris. Le côté droit de chaque partie montre le nouveau flot f résultant de l'ajout de f_p à f . Le réseau résiduel de (a) est le réseau d'entrée G . (e) Le réseau résiduel au moment du dernier test de boucle **tant que**. Il ne contient aucun chemin améliorant et le flot f montré en (d) est donc un flot maximum.

Le travail effectué dans la boucle **tant que** peut être accéléré si l'on gère efficacement la structure de données utilisée pour implémenter le réseau $G = (S, A)$. Admettons que nous gérons une structure de données correspondant à un graphe orienté $G' = (S, A')$, où $A' = \{(u, v) : (u, v) \in A \text{ ou } (v, u) \in A\}$. Les arcs du réseau G sont également des arcs de G' , et il est donc simple de gérer capacités et flux dans cette structure de données. Étant donné un flot f de G , les arcs du réseau résiduel G_f sont constitués de tous les arcs (u, v) de G' tels que $c(u, v) - f[u, v] \neq 0$. Le temps

nécessaire pour trouver un chemin dans un réseau résiduel est donc $O(S+A') = O(A)$, que l'on utilise la recherche en profondeur ou la recherche en largeur. Chaque itération de la boucle **tant que** prend donc un temps $O(A)$, ce qui donne un temps d'exécution total de $O(A |f^*|)$ pour la procédure FORD-FULKERSON.

Lorsque les capacités sont entières et que la valeur de flot optimale $|f^*|$ est faible, le temps d'exécution de l'algorithme de Ford-Fulkerson est bon. La figure 26.6(a) montre un exemple de ce qui pourrait arriver sur un réseau de transport simple pour lequel $|f^*|$ est grand. Un flot maximum dans ce réseau a la valeur 2 000 000 : 1 000 000 d'unités de transport traversent le chemin $s \rightarrow u \rightarrow t$ et 1 000 000 d'autres unités traversent le chemin $s \rightarrow v \rightarrow t$. Si le premier chemin améliorant trouvé par FORD-FULKERSON est $s \rightarrow u \rightarrow v \rightarrow t$, comme dans la figure 26.6(a), le flot prend la valeur 1 après la première itération. Le réseau résiduel résultant apparaît à la figure 26.6(b). Si la deuxième itération trouve le chemin améliorant $s \rightarrow v \rightarrow u \rightarrow t$, comme dans la figure 26.6(b), le flot prend alors la valeur 2. Le réseau résiduel qui en résulte est montré à la figure 26.6(c). On peut continuer ainsi, en choisissant le chemin améliorant $s \rightarrow u \rightarrow v \rightarrow t$ lors des itérations impaires et le chemin améliorant $s \rightarrow v \rightarrow u \rightarrow t$ lors des itérations paires. Les améliorations successives seraient alors au nombre de 2 000 000, augmentant la valeur du flot d'une seule unité à chaque fois.

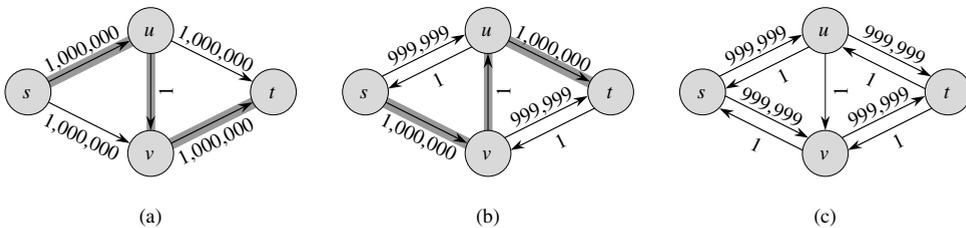


Figure 26.6 (a) Un réseau de transport pour lequel FORD-FULKERSON peut prendre un temps en $\Theta(A |f^*|)$, où f^* est un flot maximum ; on a ici $|f^*| = 2\,000\,000$. On peut voir un chemin améliorant de capacité résiduelle égale à 1. (b) Le réseau résiduel résultant. On peut voir un autre chemin améliorant de capacité résiduelle égale à 1. (c) Le réseau résiduel résultant.

f) Algorithme d'Edmonds-Karp

La borne de FORD-FULKERSON peut être améliorée si l'on implémente le calcul du chemin améliorant p en ligne 4 via une recherche en largeur, c'est-à-dire si le chemin améliorant est un *plus court* chemin de s vers t dans le réseau résiduel, où chaque arc possède une distance (pondération) unitaire. Cette implémentation particulière de la méthode de Ford-Fulkerson a pour nom **algorithme d'Edmonds-Karp**. Nous allons maintenant démontrer que l'algorithme de Edmonds-Karp s'exécute en temps $O(SA^2)$.

L'analyse dépend des distances aux sommets dans le réseau résiduel G_f . Le lemme suivant fait appel à la notation $\delta_f(u, v)$ pour représenter la distance d'un plus court chemin de u à v dans G_f , où chaque arc possède une distance unitaire.

Lemme 26.8 *Si l'algorithme de Edmonds-Karp est exécuté sur un réseau de transport $G = (S, A)$ de source s et de puits t , alors pour tous les sommets $v \in S - \{s, t\}$, la distance de plus court chemin $\delta_f(s, v)$ dans le réseau résiduel G_f augmente de façon monotone avec chaque augmentation de flot.*

Démonstration : Raisonnons par l'absurde : on suppose que, pour un certain sommet $v \in S - \{s, t\}$, il existe une augmentation de flot qui provoque la diminution de la distance de plus court chemin entre s et v . Soit f le flot juste avant la première augmentation qui diminue une certaine distance de plus court chemin, et soit f' le flot juste après. Soit v le sommet ayant le $\delta_{f'}(s, v)$ minimal dont la distance a été diminuée par l'augmentation, de sorte que $\delta_{f'}(s, v) < \delta_f(s, v)$. Soit $p = s \rightsquigarrow u \rightarrow v$ un plus court chemin de s à v dans $G_{f'}$, de sorte que $(u, v) \in A_{f'}$ et

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1. \quad (26.7)$$

Compte tenu de la façon dont on a choisi v , on sait que l'étiquette distance du sommet u n'a pas diminué, c'est-à-dire que

$$\delta_{f'}(s, u) \geq \delta_f(s, u). \quad (26.8)$$

Nous affirmons que $(u, v) \notin A_f$. Pourquoi ? Si l'on avait $(u, v) \in A_f$, alors on aurait aussi

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(d'après le lemme 24.10, relatif à l'inégalité triangulaire)} \\ &\leq \delta_{f'}(s, u) + 1 && \text{(d'après l'inégalité (26.8))} \\ &= \delta_{f'}(s, v) && \text{(d'après l'équation (26.7)) ,} \end{aligned}$$

ce qui contredit notre hypothèse que $\delta_{f'}(s, v) < \delta_f(s, v)$.

Comment peut-on avoir $(u, v) \notin A_f$ et $(u, v) \in A_{f'}$? L'augmentation doit avoir accru le flot entre v et u . L'algorithme d'Edmonds-Karp augmente toujours le flot sur des plus courts chemins, et donc le plus court chemin de s à u dans G_f a (v, u) pour dernier arc. Par conséquent,

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 && \text{(d'après l'inégalité (26.8))} \\ &= \delta_{f'}(s, v) - 2 && \text{(d'après l'équation (26.7)) ,} \end{aligned}$$

ce qui contredit notre hypothèse que $\delta_{f'}(s, v) < \delta_f(s, v)$. On en conclut que notre hypothèse selon laquelle il existe un tel sommet v est erronée. \square

Le théorème suivant borne le nombre d'itérations de l'algorithme d'Edmonds-Karp.

Théorème 26.9 *Si l'algorithme d'Edmonds-Karp est exécuté sur un réseau de transport $G = (S, A)$ de source s et de puits t , alors le nombre total d'augmentations de flot effectuées par l'algorithme est $O(SA)$.*

Démonstration : On dit qu'un arc (u, v) d'un réseau résiduel G_f est **critique** sur un chemin améliorant p si la capacité résiduelle de p est la capacité résiduelle de (u, v) , c'est-à-dire si $c_f(p) = c_f(u, v)$. Après que nous avons augmenté le flot le long d'un chemin améliorant, tout arc critique du chemin disparaît du réseau résiduel. De plus, il faut qu'un arc au moins soit critique dans chaque chemin améliorant. Nous allons montrer que chacun des $|A|$ arcs peut devenir critique au plus $|S|/2 - 1$ fois.

Soit u et v deux sommets de S reliés par un arc de A . Puisque les chemins améliorants sont des plus courts chemins, quand (u, v) est critique pour la première fois, on a

$$\delta_f(s, v) = \delta_f(s, u) + 1 .$$

Après augmentation du flot, l'arc (u, v) disparaît du réseau résiduel. Il ne pourra pas réapparaître sur un autre chemin améliorant tant que le flux de u à v n'aura pas diminué, ce qui n'arrive que si (v, u) apparaît sur un chemin améliorant. Si f' est le flot de G au moment de cet événement, on a

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 .$$

Comme $\delta_f(s, v) \leq \delta_{f'}(s, v)$ d'après le lemme 26.8, on a

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 . \end{aligned}$$

En conséquence, entre le moment où (u, v) devient critique et celui où il devient à nouveau critique, la distance entre u et la source augmente au moins de 2. La distance entre u et la source est initialement au moins égale à 0. Les sommets intermédiaires d'un plus court chemin de s à u ne peuvent pas contenir s , u ou t (car (u, v) sur le chemin critique entraîne que $u \neq t$). Donc, jusqu'à ce que u devienne éventuellement inaccessible à partir de la source, sa distance est au plus égale à $|S| - 2$. Donc, (u, v) peut devenir critique au plus $(|S| - 2)/2 = |S|/2 - 1$ fois. Puisqu'il existe $O(A)$ paires de sommets pouvant être reliés par un arc dans un graphe résiduel, le nombre total d'arcs critiques pendant toute l'exécution de l'algorithme d'Edmonds-Karp est $O(SA)$. Chaque chemin améliorant comporte au moins un arc critique, et le théorème en découle. \square

Comme chaque itération de FORD-FULKERSON peut s'implémenter en temps $O(A)$ quand le chemin améliorant est trouvé par une recherche en largeur, le temps d'exécution total de l'algorithme d'Edmonds-Karp est $O(SA^2)$. Nous verrons que les algorithmes de préflot peuvent donner des bornes encore meilleures. L'algorithme de la section 26.4 donne une méthode dont le temps d'exécution est $O(S^2A)$, qui forme la base de l'algorithme à temps $O(S^3)$ étudié à la section 26.5.

Exercices

26.2.1 Dans la figure 26.1(b), donner la valeur du flot traversant la coupe $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? Quelle est la capacité de cette coupe ?

26.2.2 Montrer l'exécution de l'algorithme de Edmonds-Karp sur le réseau de transport de la figure 26.1(a).

26.2.3 Dans l'exemple de la figure 26.5, quelle est la coupe minimum correspondant au flot maximum représenté ? Parmi les chemins améliorants qui apparaissent dans cet exemple, quels sont les deux qui annulent des flux ?

26.2.4 Démontrer que, pour tout couple de sommets u et v , pour toute fonction de capacité c et pour toute fonction de flot f , on a $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

26.2.5 On se souvient de ce que la construction vue à la section 26.1 qui convertit un réseau de transport à sources et puits multiples en un réseau à source et puits uniques ajoute des arcs de capacité infinie. Démontrer qu'un flot quelconque du réseau résultant possède une valeur finie si les arcs du réseau initial à sources et puits multiples ont des capacités finies.

26.2.6 Supposons que chaque source s_i d'un problème à sources et puits multiples produise exactement p_i unités de flot, de sorte que $f(s_i, S) = p_i$. Supposons par ailleurs que chaque puits t_j consomme exactement q_j unités de flot, de sorte que $f(V, t_j) = q_j$, où $\sum_i p_i = \sum_j q_j$. Montrer comment convertir le problème consistant à trouver un flot f qui respecte ces contraintes supplémentaires en un problème consistant à trouver un flot maximum dans un réseau de transport à source et puits uniques.

26.2.7 Démontrer le lemme 26.3.

26.2.8 Montrer qu'on peut toujours trouver un flot maximum dans un réseau $G = (S, A)$ à l'aide d'une séquence d'au plus $|A|$ chemins améliorants. (*Conseil* : Déterminer les chemins après avoir trouvé le flot maximum.)

26.2.9 La *connectivité d'arête* d'un graphe non orienté est le nombre minimal k d'arêtes qu'il faut supprimer pour que le graphe ne soit plus connexe. Par exemple, la connectivité d'arc d'un arbre est 1 et celle d'un cycle de sommets est 2. Montrer comment déterminer la connectivité d'arête d'un graphe non orienté $G = (S, A)$ en exécutant un algorithme de flot maximum sur au plus $|S|$ réseaux de transport, comportant chacun $O(S)$ sommets et $O(A)$ arêtes.

26.2.10 Supposons qu'un réseau de transport $G = (S, A)$ comporte des arcs symétriques, c'est-à-dire que $(u, v) \in A$ si et seulement si $(v, u) \in A$. Montrer que l'algorithme d'Edmonds-Karp se termine après au plus $|S| |A| / 4$ itérations. (*conseil* : Pour un arc (u, v) quelconque, regarder comment $\delta(s, u)$ et $\delta(v, t)$ évoluent entre deux instants où (u, v) est critique.)

26.3 COUPLAGE MAXIMUM DANS UN GRAPHE BIPARTI

Certains problèmes combinatoires peuvent aisément se ramener à un problème de flot maximum. Le problème du flot maximum à sources et puits multiples étudié à la section 26.1 nous en a donné un exemple. Il y a d'autres problèmes combinatoires, qui peuvent sembler à priori n'avoir que peu de rapport avec les réseaux de transport, mais qui peuvent en fait se ramener à des problèmes de flot maximum. Cette section présente un tel problème : trouver un couplage maximum dans un graphe biparti (voir section B.4). Pour résoudre ce problème, nous nous baserons sur une propriété d'intégralité fournie par la méthode de Ford-Fulkerson. Nous verrons également que la méthode de Ford-Fulkerson peut être adaptée pour résoudre le problème du couplage biparti maximum sur un graphe $G = (S, A)$ en temps $O(SA)$.

a) Problème du couplage biparti maximum

Étant donné un graphe non orienté $G = (S, A)$, un **couplage** est un sous-ensemble d'arête $M \subseteq A$ tel que pour tous les sommets $v \in S$, au plus une arête de M est incidente à v . On dit qu'un sommet $v \in S$ est **couvert** par le couplage M si une certaine arête de M est incidente à v ; autrement, on dit que v n'est **pas couvert**. Un **couplage maximum** est un couplage de cardinalité maximum, c'est-à-dire un couplage M tel que, pour tout couplage M' , on ait $|M| \geq |M'|$. Dans cette section, nous nous bornons à rechercher les couplages maxima dans des graphes bipartis. On suppose que l'ensemble des sommets peut être partitionné en $S = L \cup R$, où L et R sont disjoints et où toutes les arêtes de A passent entre L et R . On suppose par ailleurs que chaque sommet de S a au moins une arête incidente. La figure 26.7 illustre la notion de couplage.

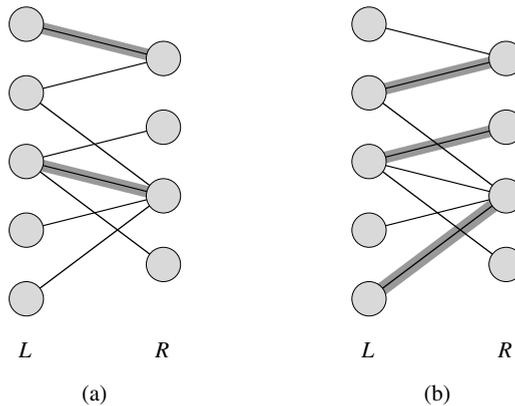


Figure 26.7 Un graphe biparti $G = (S, A)$ avec une partition des sommets $S = L \cup R$. (a) Un couplage de cardinalité 2. (b) Un couplage maximum de cardinalité 3.

Le problème consistant à trouver un couplage maximum dans un graphe biparti trouve de nombreuses applications pratiques. Par exemple, on pourrait coupler un ensemble L de machines avec un ensemble R de tâches à effectuer simultanément. On considère la présence d'une arête (u, v) dans A comme signifiant qu'une machine particulière $u \in L$ est capable d'effectuer une tâche particulière $v \in R$. Un couplage maximum permet de faire travailler le plus de machines possibles.

b) Recherche d'un couplage biparti maximum

La méthode de Ford-Fulkerson permet de trouver un couplage maximum dans un graphe biparti $G = (S, A)$ non orienté, avec un temps polynomial en $|S|$ et $|A|$. L'astuce est de construire un réseau de transport dans lequel les flots correspondent aux couplages, comme le montre la figure 26.8. On définit le **réseau de transport correspondant** $G' = (S', A')$ pour le graphe biparti G de la manière suivante. Soit la source s

et le puits t deux nouveaux sommets n'appartenant pas à S , et soit $S' = S \cup \{s, t\}$. Si la partition des sommets de G est $S = L \cup R$, les arcs orientés de G' sont les arêtes de A , orientées de L vers R , plus $|S|$ nouveaux arcs :

$$\begin{aligned} A' &= \{(s, u) : u \in L\} \\ &\cup \{(u, v) : u \in L, v \in R, \text{ et } (u, v) \in A\} \\ &\cup \{(v, t) : v \in R\} . \end{aligned}$$

Pour compléter la construction, on affecte une capacité unitaire à chaque arc de A' . Comme chaque sommet de S a au moins un arc incident, $|A| \geq |S|/2$. Donc, $|A| \leq |A'| = |A| + |S| \leq 3|A|$, et donc $|A'| = \Theta(|A|)$.

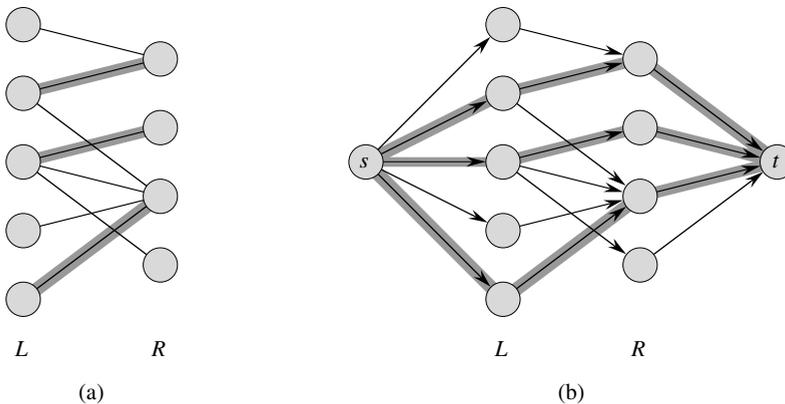


Figure 26.8 Réseau de transport correspondant à un graphe biparti. (a) Le graphe biparti $G = (S, A)$ avec la partition de sommets $S = L \cup R$ de la figure 26.7. Un couplage maximum est représenté par des arêtes sur fond ombré. (b) Le réseau de transport G' correspondant, avec un flot maximum affiché. Chaque arc possède une capacité unitaire. Les arcs sur fond ombré ont un flux égal à 1 et tous les autres arcs ne transportent aucun flux. Les arcs ombrés de L vers R correspondent à ceux d'un couplage maximum du graphe biparti.

Le lemme suivant montre qu'un couplage de G correspond directement à un flot de G' , réseau de transport correspondant de G . On dit qu'un flot f d'un réseau de transport $G = (S, A)$ est à **valeurs entières** si $f(u, v)$ est un entier pour tout $(u, v) \in S \times S$.

Lemme 26.10 Soit $G = (S, A)$ un graphe biparti avec une partition de sommets $V = L \cup R$ et soit $G' = (S', A')$ son réseau de transport correspondant. Si M est un couplage de G , alors il existe un flot à valeurs entières f dans G' , avec $|f| = |M|$. Inversement, si f est un flot à valeurs entières de G' , alors il existe un couplage M dans G de cardinalité $|M| = |f|$.

Démonstration : Commençons par démontrer qu'un couplage M de G correspond à un flot à valeurs entières f de G' . On définit f de la manière suivante. Si $(u, v) \in M$, alors $f(s, u) = f(u, v) = f(v, t) = 1$ et $f(u, s) = f(v, u) = f(t, v) = -1$. Pour tous les

autres arcs $(u, v) \in A'$, on définit $f(u, v) = 0$. Il est immédiat de vérifier que f vérifie les propriétés de symétrie, de contraintes de capacité et de conservation de flot.

Intuitivement, chaque arc $(u, v) \in M$ correspond à 1 unité de flot dans G' qui traverse le chemin $s \rightarrow u \rightarrow v \rightarrow t$. Par ailleurs, les chemins construits à partir des arêtes de M n'ont aucun sommet commun, exceptés s et t . Le flot net à travers la coupe $(L \cup \{s\}, R \cup \{t\})$ est égal à $|M|$; donc, d'après le lemme 26.5, la valeur du flot est $|f| = |M|$. Pour prouver la réciproque, soit f un flot à valeurs entières de G' et soit

$$M = \{(u, v) : u \in L, v \in R, \text{ et } f(u, v) > 0\} .$$

Chaque sommet $u \in L$ ne possède qu'un seul arc entrant, à savoir (s, u) , et sa capacité est 1. Donc, chaque $u \in L$ voit au plus une unité de flux positif arriver jusqu'à lui; si effectivement une unité de flux positif arrive en u , le principe de conservation du flot dit qu'une unité de flux positif doit partir de u . En outre, comme f est à valeurs entières, pour chaque $u \in L$, l'unique unité de flux positif qui arrive éventuellement en u y arrive *via* un seul arc et la quantité équivalente part de u *via* un seul arc. Donc, u reçoit une et une seule unité de flux positif si et seulement si il y a un et un seul sommet $v \in R$ tel que $f(u, v) = 1$, et il y a au plus un arc partant de chaque $u \in L$ qui transporte un flux positif. On raisonne de manière symétrique pour chaque $v \in R$. L'ensemble M est donc un couplage.

Pour voir que $|M| = |f|$, on remarque que, pour tout sommet $u \in L$ couvert, on a $f(s, u) = 1$ et que, pour toute arête $(u, v) \in A - M$, on a $f(u, v) = 0$. Par suite,

$$\begin{aligned} |M| &= f(L, R) \\ &= f(L, S') - f(L, L) - f(L, s) - f(L, t) \quad (\text{d'après le lemme 26.1}) . \end{aligned}$$

On peut considérablement simplifier cette expression. La conservation du flot implique que $f(L, S') = 0$; le lemme 26.1 implique que $f(L, L) = 0$; la symétrie implique que $-f(L, s) = f(s, L)$; et comme il n'y a pas d'arcs entre L et t , on a $f(L, t) = 0$. D'où,

$$\begin{aligned} |M| &= f(s, L) \\ &= f(s, S') \quad (\text{car tous les arcs partant de } s \text{ vont vers } L) \\ &= |f| \quad (\text{d'après la définition de } |f|) . \quad \square \end{aligned}$$

En se basant sur le lemme 26.10, on pourrait penser qu'un couplage maximum d'un graphe biparti correspond à un flot maximum du réseau de transport correspondant G' , et que l'on peut donc calculer un couplage maximum de G en exécutant un algorithme de flot maximum sur G' . Le seul hic dans ce raisonnement, c'est que l'algorithme de flot maximum peut retourner un flot de G' pour lequel un certain $f(u, v)$ n'est pas un entier, même si la valeur de flot $|f|$ est forcément entière. Le théorème suivant montre que, si l'on emploie la méthode de Ford-Fulkerson, ce problème ne se pose pas.

Théorème 26.11 (Théorème de l'intégralité) *Si la fonction de capacité c ne prend que des valeurs entières, alors le flot maximum f produit par la méthode de Ford-Fulkerson a pour propriété que $|f|$ est à valeur entière. De plus, pour tous sommets u et v , la valeur de $f(u, v)$ est un entier.*

Démonstration : La démonstration se fait par récurrence sur le nombre d'itérations. Elle est laissée en exercice (voir exercice 26.3.2). \square

On peut montrer à présent un corollaire du lemme 26.10.

Corollaire 26.12 *La cardinalité d'un couplage maximum M d'un graphe biparti G est la valeur d'un flot maximum du réseau de transport correspondant G' .*

Démonstration : On utilise la terminologie du lemme 26.10. Supposons que M soit un couplage maximum de G et que le flot correspondant f dans G' ne soit pas maximum. Alors il existe un flot maximum f' de G' tel que $|f'| > |f|$. Comme les capacités de G' sont à valeurs entières, d'après le théorème 26.11, on peut supposer que f' est entier. Donc, f' correspond à un couplage M' de G de cardinalité $|M'| = |f'| > |f| = |M|$, ce qui contredit notre hypothèse selon laquelle M est un couplage maximum. De même, on peut montrer que si f est un flot maximum de G' , son couplage correspondant est un couplage maximum de G . \square

Donc, étant donné un graphe non orienté biparti G , on peut trouver un couplage maximum en créant le réseau de transport G' , en exécutant la méthode de Ford-Fulkerson, puis en obtenant directement un couplage maximum M à partir du flot maximum à valeur entières f trouvé. Puisqu'un couplage quelconque d'un graphe biparti possède un cardinal qui est au plus égal à $\min(L, R) = O(S)$, la valeur du flot maximum de G' est $O(S)$. On peut donc trouver un couplage maximum dans un graphe biparti en un temps $O(SA') = O(SA)$, car $|A'| = \Theta(A)$.

Exercices

26.3.1 Exécuter l'algorithme de Ford-Fulkerson sur le réseau de transport de la figure 26.8(b) et montrer le réseau résiduel après chaque augmentation de flot. Numéroter les sommets de L de 1 à 5 du haut vers le bas et ceux de R de 6 à 9 du haut vers le bas. Pour chaque itération, choisir le chemin améliorant qui est le plus petit lexicographiquement.

26.3.2 Démontrer le théorème 26.11.

26.3.3 Soit $G = (S, A)$ un graphe biparti ayant la partition de sommets $S = L \cup R$, et soit G' son réseau de transport correspondant. Donner un bon majorant de la longueur d'un chemin améliorant quelconque trouvé dans G' pendant l'exécution de FORD-FULKERSON.

26.3.4 * Un **couplage parfait** est un couplage dans lequel chaque sommet est couvert. Soit $G = (S, A)$ un graphe biparti non orienté ayant la partition de sommets $S = L \cup R$, où $|L| = |R|$. Pour un $X \subseteq S$ quelconque, on définit le **voisinage** de X par

$$V(X) = \{y \in S : (x, y) \in A \text{ pour un certain } x \in X\},$$

c'est-à-dire l'ensemble des sommets adjacents à un certain membre de X . Démontrer le **théorème de Hall** : il existe un couplage parfait dans G si et seulement si $|E| \leq |V(E)|$ pour tout sous-ensemble $E \subseteq L$.

26.3.5 * Un graphe biparti $G = (S, A)$, où $S = L \cup R$, est **d -régulier** si tout sommet $v \in S$ a un degré exactement égal à d . Pour tout graphe biparti d -régulier, on a $|L| = |R|$. Démontrer que tout graphe biparti d -régulier possède un couplage de cardinalité $|L|$, en montrant qu'une coupe minimum du réseau de transport correspondant a une capacité égale à $|L|$.

26.4 [★] ALGORITHMES DE PRÉFLOTS

Dans cette section, nous présentons l'approche « pousser-réétiqueter » du calcul des flots maximum. Les algorithmes de flot maximum parmi les plus rapides connus à ce jour sont des algorithmes de préflots et d'autres problèmes de flot, tel le problème du flot de coût minimal, peuvent être efficacement résolus par des méthodes de préflots. Cette section présentera l'algorithme « générique » de Goldberg pour la recherche d'un flot maximum, dont une implémentation simple s'exécute en temps $O(S^2A)$, ce qui améliore la borne $O(SA^2)$ de l'algorithme d'Edmonds-Karp. La section 26.5 affina l'algorithme générique pour obtenir un autre algorithme de préflots qui s'exécute en temps $O(S^3)$.

Les algorithmes de préflots travaillent de façon plus localisée que la méthode de Ford-Fulkerson. Au lieu d'examiner le réseau résiduel tout entier à la recherche d'un chemin améliorant, ils se concentrent sur un sommet à la fois en ne s'intéressant qu'à ses voisins dans le réseau résiduel. Par ailleurs, contrairement à la méthode de Ford-Fulkerson, les algorithmes de préflots ne tentent pas de maintenir la propriété de conservation du flot tout au long de l'exécution. Néanmoins, ils maintiennent un **préflot**, qui est une fonction $f : S \times S \rightarrow \mathbf{R}$ qui satisfait à la propriété de symétrie, aux contraintes de capacité, ainsi qu'à la version relâchée suivante de la conservation de flot : $f(S, u) \geq 0$ pour tout sommet $u \in S - \{s\}$. On appelle cette quantité l'**excédent de flot** sur u , donné par

$$e(u) = f(S, u) . \quad (26.9)$$

On dit qu'un sommet $u \in S - \{s, t\}$ **déborde** si $e(u) > 0$.

Nous commencerons cette section en décrivant les concepts intuitifs sous-jacents à la méthode de préflots. Nous étudierons ensuite les deux opérations utilisées par la méthode : « poussage » et « ré-étiquetage » d'un sommet. Enfin, nous présenterons un algorithme de préflots générique dont nous analyserons la validité et le temps d'exécution.

a) Intuition

Le principe de la méthode de préflots se comprend probablement mieux si l'on prend l'exemple d'un flot de liquide : on considère un réseau de transport $G = (S, A)$ comme un système de tuyaux de capacités fixées et reliés entre eux. En appliquant cette analogie à la méthode de Ford-Fulkerson, on pourrait dire qu'un chemin améliorant du réseau permet de faire passer un écoulement supplémentaire, sans points de branchement, entre la source et le puits. La méthode de Ford-Fulkerson fait passer à chaque itération un peu plus de liquide dans les tuyaux, jusqu'à ce qu'il ne puissent pas en contenir plus.

L'algorithme de préflots générique ne procède pas de la même façon. Comme précédemment, les arcs correspondent à des tuyaux. Les sommets, qui sont des raccords, possèdent deux propriétés intéressantes. Premièrement, pour évacuer le flot excédentaire, chaque sommet est doté d'un tuyau de vidange, menant à un réservoir arbitrairement grand pouvant accumuler le liquide.

Deuxièmement, chaque sommet, son réservoir et tous ses raccordements se trouvent sur une plate-forme dont la hauteur croît à mesure que l'algorithme progresse.

Les hauteurs des sommets déterminent la façon dont le flot est poussé : on ne pousse que vers le bas, c'est-à-dire d'un sommet supérieur vers un sommet inférieur. Il peut exister un flux positif entre un sommet inférieur et un sommet supérieur, mais les opérations de poussage agissent toujours de haut en bas. La hauteur de la source est fixée à $|S|$ et celle du puits est fixée à 0. Tous les autres sommets commencent à la hauteur 0 et montent avec le temps. L'algorithme commence par faire descendre le plus de flot possible depuis la source vers le puits. La quantité envoyée est juste assez importante pour remplir au maximum chaque tuyau partant de la source ; autrement dit, il envoie la capacité de la coupe $(s, S - s)$. Quand le flot arrive sur un sommet intermédiaire, il est récupéré dans le réservoir du sommet. De là, il sera ensuite poussé vers le bas.

Il se peut que les seuls tuyaux qui partent d'un sommet u sans être déjà saturés soient reliés à l'autre bout à des sommets qui se trouvent sur un niveau égal ou supérieurs à celui de u . Dans ce cas, pour débarrasser un sommet u débordant de son flot excédentaire, il faut augmenter sa hauteur, opération appelée « ré-étiquetage » du sommet u . Sa hauteur est augmentée d'une unité de plus que la hauteur du plus bas de ses voisins vers lequel il possède un tuyau non saturé. Après qu'un sommet a été réétiqueté, il existe donc au moins un tuyau sortant par lequel on peut pousser plus de flot.

Avec ce système, tout le flot pouvant arriver jusqu'au puits finit par arriver à destination. Il ne peut y en avoir plus, puisque les tuyaux obéissent aux contraintes de capacité ; la quantité de flot traversant une coupe quelconque reste limitée par la capacité de la coupe. Pour que le préflot se transforme en flot « légal », l'algorithme renvoie ensuite à la source l'excédent contenu dans les réservoirs des sommets débordants, en continuant à ré-étiqueter les sommets pour qu'ils dépassent la hauteur $|S|$ fixée de la source. Comme nous le verrons, une fois que tous les réservoirs ont été vidés, le préflot n'est pas seulement un flot « légal » mais aussi un flot maximum.

b) Opérations élémentaires

La discussion précédente montre qu'il existe deux opérations élémentaires effectuées par un algorithme de préflot : le poussage du flot excédentaire depuis un sommet vers l'un de ses voisins et le ré-étiquetage d'un sommet. Les conditions d'application de ces opérations dépendent des hauteurs des sommets, que nous allons à présent définir précisément.

Soit $G = (S, A)$ un réseau de transport de source s et de puits t et soit f un préflot de G . Une fonction $h : S \rightarrow \mathbf{N}$ est une **fonction de hauteur** ⁽³⁾ si $h(s) = |S|$, $h(t) = 0$ et

$$h(u) \leq h(v) + 1$$

pour tout arc résiduel $(u, v) \in A_f$. On obtient immédiatement le lemme suivant.

(3) Dans la littérature, une fonction de hauteur porte généralement le nom de « fonction de distance » et la hauteur d'un sommet est dite « étiquette de distance ». Nous employons le terme « hauteur », car il reflète mieux les intuitions sous-jacentes à l'algorithme. Nous conservons le terme « ré-étiqueter » pour désigner l'opération qui accroît la hauteur d'un sommet. La hauteur d'un sommet est liée à sa distance par rapport au puits t , telle que donnée par une recherche en largeur dans le transposé G^T .

Lemme 26.13 Soit $G = (S, A)$ un réseau de transport, soit f un préflot de G et soit h une fonction de hauteur sur S . Pour deux sommets $u, v \in S$ quelconques, si $h(u) > h(v) + 1$, alors (u, v) n'est pas un arc du graphe résiduel.

L'opération élémentaire $\text{POUSSER}(u, v)$ peut s'appliquer si u est un sommet débordant, $c_f(u, v) > 0$ et $h(u) = h(v) + 1$. Le pseudo code suivant met à jour le préflot f d'un réseau concerné $G = (S, A)$. Il suppose que les capacités résiduelles peuvent aussi être calculées en temps constant, connaissant c et f . Le flot excédentaire conservé dans un sommet u est représenté par l'attribut $e[u]$ et la hauteur de u est représentée par l'attribut $h[u]$. L'expression $d_f(u, v)$ est une variable temporaire qui stocke la quantité de flot pouvant être poussée de u vers v .

$\text{POUSSER}(u, v)$

- 1 \triangleright **s'applique quand** : u déborde, $c_f[u, v] > 0$ et $h[u] = h[v] + 1$.
- 2 \triangleright **action** : pousser $d_f(u, v) = \min(e[u], c_f(u, v))$ unités de flot de u vers v .
- 3 $d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$
- 4 $f[u, v] \leftarrow f[u, v] + d_f(u, v)$
- 5 $f[v, u] \leftarrow -f[v, u]$
- 6 $e[u] \leftarrow e[u] - d_f(u, v)$
- 7 $e[v] \leftarrow e[v] + d_f(u, v)$

Le code de POUSSER agit comme suit. On suppose que le sommet u possède un excédent positif $e[u]$ et que la capacité résiduelle de (u, v) est positive. Il est donc possible d'accroître le flot entre u et v d'une quantité $d_f(u, v) = \min(e[u], c_f(u, v))$, sans que $e[u]$ devienne négatif ni que soit dépassée la capacité $c(u, v)$. La ligne 3 calcule la valeur $d_f(u, v)$, et on actualise f aux lignes 4–5 et e aux lignes 6–7. Donc, si f est un préflot avant que POUSSER soit appelé, il reste un préflot après.

On remarque que rien dans le code de POUSSER ne dépend des hauteurs de u et v , bien qu'on interdise son exécution tant qu'on n'a pas $h[u] = h[v] + 1$. Donc, le flot excédentaire n'est poussé vers le bas que si la différence de hauteur est égale à 1. D'après le lemme 26.13, aucun arc résiduel n'existe entre deux sommets dont les hauteurs diffèrent de plus de 1 et donc, du moment que l'attribut h est bien une fonction de hauteur, il n'y a rien à gagner en permettant au flot d'être poussé vers le bas depuis un dénivelé supérieur à 1.

On appelle l'opération $\text{POUSSER}(u, v)$ un *poussage* de u vers v . Si un poussage s'applique à un certain arc (u, v) partant du sommet u , on dit aussi qu'elle s'applique à u . C'est un *poussage saturant* si l'arc (u, v) devient *saturé* ($c_f(u, v) = 0$ après poussage); autrement, c'est un *poussage non saturant*. Si un arc est saturé, il n'apparaît pas dans le réseau résiduel. Un lemme simple caractérise un résultat d'un poussage non saturant.

Lemme 26.14 Après un poussage non saturant de u vers v , le sommet u n'est plus débordant.

Démonstration : Comme le poussage était non saturant, la quantité de flot $d_f(u, v)$ affectivement poussée doit être égale à $e[u]$ avant le poussage. Comme $e[u]$ est diminué de cette quantité, il devient 0 après le poussage. \square

L'opération élémentaire RÉÉTIQUETER(u) s'applique si u déborde et si $h[u] \leq h[v]$ pour tout arc $(u, v) \in A_f$. Autrement dit, on peut ré-étiqueter un sommet u débordant si, pour tout sommet v pour lequel il existe une capacité résiduelle de u vers v , le flot ne peut pas être poussé de u vers v car v n'est pas plus bas que u . (Ne pas oublier que, par définition, ni la source s ni le puits t ne peuvent déborder et donc que ni l'une ni l'autre ne peuvent être réétiquetés.)

RÉÉTIQUETER(u)

- 1 \triangleright **s'applique quand :** u est débordant et, pour tout $v \in S$ tel que $(u, v) \in A_f$, on a $h[u] \leq h[v]$.
- 2 \triangleright **action :** accroît la hauteur de u .
- 3 $h[u] \leftarrow 1 + \min \{h[v] : (u, v) \in A_f\}$

Lorsqu'on appelle l'opération RÉÉTIQUETER(u), on dit que le sommet u est **rééti-queté**. Notez que, quand u est rééti-queté, A_f doit contenir au moins un arc sortant de u pour que la minimisation effectuée dans le code s'applique à un ensemble non vide. Cette propriété provient de l'hypothèse selon laquelle u déborde. Comme $e[u] > 0$, on a $e[u] = f[S, u] > 0$ et il faut donc qu'il existe au moins un sommet v tel que $f[v, u] > 0$. Mais alors

$$\begin{aligned} c_f(u, v) &= c(u, v) - f[u, v] \\ &= c(u, v) + f[v, u] \\ &> 0, \end{aligned}$$

ce qui implique que $(u, v) \in A_f$. L'opération RÉÉTIQUETER(u) donne donc à u la plus grande hauteur permise par les contraintes de la fonction de hauteur.

c) Algorithme générique

L'algorithme de préflots générique fait appel au sous-programme suivant pour créer un préflot initial dans le réseau de transport.

INITIALISER-PRÉFLOT(G, s)

- 1 **pour** chaque sommet $u \in S[G]$
- 2 **faire** $h[u] \leftarrow 0$
- 3 $e[u] \leftarrow 0$
- 4 **pour** chaque arc $(u, v) \in A[G]$
- 5 **faire** $f[u, v] \leftarrow 0$
- 6 $f[v, u] \leftarrow 0$
- 7 $h[s] \leftarrow |S[G]|$
- 8 **pour** chaque sommet $u \in Adj[s]$
- 9 **faire** $f[s, u] \leftarrow c(s, u)$
- 10 $f[u, s] \leftarrow -c(s, u)$
- 11 $e[u] \leftarrow c(s, u)$
- 12 $e[s] \leftarrow e[s] - c(s, u)$

Cette procédure crée un préflot f initial, défini par

$$f[u, v] = \begin{cases} c(u, v) & \text{si } u = s, \\ -c(v, u) & \text{si } v = s, \\ 0 & \text{sinon.} \end{cases} \quad (26.10)$$

Autrement dit, chaque arc partant de la source est rempli au maximum et tous les autres arcs ne transportent aucun flux. Pour chaque sommet v adjacent à la source, on a initialement $e[v] = c(s, v)$ et $e[s]$ est initialisé à une valeur qui est l'opposé de la somme de ces capacités. L'algorithme générique démarre également avec une fonction de hauteur h initiale, donnée par

$$h[u] = \begin{cases} |S| & \text{si } u = s, \\ 0 & \text{sinon.} \end{cases}$$

C'est une fonction de hauteur car les seuls arcs (u, v) pour lesquels $h[u] > h[v] + 1$ sont ceux où $u = s$ et ces arcs sont saturés, ce qui signifie qu'ils n'appartiennent pas au réseau résiduel.

L'initialisation, suivie d'une séquence de poussages et ré-étiquetages exécutés dans un ordre quelconque, donne l'algorithme PRÉFLOT-GÉNÉRIQUE :

PRÉFLOT-GÉNÉRIQUE(G)

- 1 INITIALISER-PRÉFLOT(G, s)
- 2 **tant que** il est possible d'appliquer un poussage ou un ré-étiquetage
- 3 **faire** choisir un poussage ou ré-étiquetage applicable et l'exécuter

Le lemme suivant nous dit que, tant qu'il existe un sommet débordant, au moins une des deux opérations élémentaires peut s'appliquer.

Lemme 26.15 (Un sommet débordant peut être poussé ou réétiqueté) *Soit $G = (S, A)$ un réseau de transport de source s et de puits t , soit f un préflot et soit h une fonction de hauteur pour f . Si u est un sommet débordant, alors on peut lui appliquer soit un poussage, soit un ré-étiquetage*

Démonstration : Pour un arc résiduel (u, v) quelconque, on a $h(u) \leq h(v) + 1$ car h est une fonction de hauteur. Si une opération de poussage ne peut pas s'appliquer à u , alors pour tous les arcs résiduels (u, v) , on doit avoir $h(u) < h(v) + 1$, ce qui implique que $h(u) \leq h(v)$. Donc, une opération de ré-étiquetage peut être appliquée à u . \square

d) Validité de la méthode par préflot

Pour montrer que l'algorithme de préflot générique résout le problème du flot maximum, nous allons commencer par démontrer que, s'il se termine, le préflot f est un flot maximum. Nous prouverons plus tard qu'il se termine. Commençons par quelques observations sur la fonction de hauteur h .

Lemme 26.16 (Les hauteurs de sommet ne diminuent jamais) *Pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur un réseau de transport $G = (S, A)$, pour chaque sommet $u \in S$, la hauteur $h[u]$ ne diminue jamais. De plus, à chaque opération de ré-étiquetage appliquée à un sommet u , sa hauteur $h[u]$ croît au moins de 1.*

Démonstration : Comme les hauteurs des sommets ne sont modifiées que pendant les opérations de ré-étiquetage, il suffit de démontrer la deuxième affirmation du lemme. Si le sommet u est sur le point d'être réétiqueté, alors pour tous les sommets v tels que $(u, v) \in A_f$, on a $h[u] \leq h[v]$. Donc, que $h[u] < 1 + \min \{h[v] : (u, v) \in A_f\}$ et donc l'opération augmente nécessairement $h[u]$. \square

Lemme 26.17 *Soit $G = (S, A)$ un réseau de transport de source s et de puits t . Pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur G , l'attribut h reste en permanence une fonction de hauteur.*

Démonstration : La démonstration se fait par récurrence sur le nombre d'opérations élémentaires effectuées. Au départ, h est une fonction de hauteur, comme nous l'avons déjà vu.

Nous affirmons que, si h est une fonction de hauteur, elle le reste après une opération RÉÉTIQUETER(u). Si l'on prend un arc résiduel $(u, v) \in A_f$ sortant de u , alors l'opération RÉÉTIQUETER(u) garantit que $h[u] \leq h[v] + 1$ après. Considérons maintenant un arc résiduel (w, u) entrant dans u . D'après le lemme 26.16, $h[w] \leq h[u] + 1$ avant l'opération RÉÉTIQUETER(u) entraîne que $h[w] < h[u] + 1$ après. Donc, h reste une fonction de hauteur après l'opération RÉÉTIQUETER(u).

Considérons à présent une opération POUSSER(u, v). Cette opération peut ajouter l'arc (v, u) à A_f , et elle peut supprimer l'arc (u, v) de A_f . Dans le premier cas, on a $h[v] = h[u] - 1 < h[u] + 1$, et donc h reste une fonction de hauteur. Dans le second cas, la suppression de (u, v) dans le réseau résiduel supprime la contrainte correspondante et h reste, ici aussi, une fonction de hauteur. \square

Le lemme suivant donne une propriété importante des fonctions de hauteur.

Lemme 26.18 *Soit $G = (S, A)$ un réseau de transport de source s et de puits t , soit f un préflot de G et soit h une fonction de hauteur sur S . Alors, il n'existe aucun chemin allant de la source s au puits t dans le réseau résiduel G_f .*

Démonstration : Supposons, en raisonnant par l'absurde, qu'il existe un chemin $p = \langle v_0, v_1, \dots, v_k \rangle$ de s vers t dans G_f , où $v_0 = s$ et $v_k = t$. Sans perdre la généralité du raisonnement, on peut supposer que p est un chemin élémentaire et donc $k < |S|$. Pour $i = 0, 1, \dots, k - 1$, l'arc $(v_i, v_{i+1}) \in A_f$. Comme h est une fonction de hauteur, $h(v_i) \leq h(v_{i+1}) + 1$ pour $i = 0, 1, \dots, k - 1$. En combinant ces inégalités sur tout le chemin p , on obtient $h(s) \leq h(t) + k$. Mais comme $h(t) = 0$, on a $h(s) \leq k < |S|$, ce qui contredit la contrainte $h(s) = |S|$ pour une fonction de hauteur. \square

Nous pouvons à présent montrer que, si l'algorithme de préflot générique se termine, le préflot calculé est un flot maximum.

Théorème 26.19 (Validité de l'algorithme de préflots générique) *Si l'algorithme PRÉFLOT-GÉNÉRIQUE se termine lorsqu'il est exécuté sur un réseau de transport $G = (S, A)$ de source s et de puits t , alors le préflot f calculé est un flot maximum de G .*

Démonstration : On utilise l'invariant de boucle que voici : À chaque exécution du test de la boucle **tant que** en ligne 2 de PRÉFLOT-GÉNÉRIQUE, f est un préflot.

Initialisation : INITIALISER-PRÉFLOT fait de f un préflot.

Conservation : Les seules opérations faites dans la boucle **tant que** des lignes 2-3 sont le poussage et le ré-étiquetage. Les opérations de ré-étiquetage affectent uniquement les attributs hauteur, mais pas les valeurs du flot ; donc, elles n'affectent pas le fait que f soit un préflot ou non. Comme prouvé à la page 651, si f est un préflot avant un poussage, il reste un préflot après.

Terminaison : À la fin de l'exécution, chaque sommet de $S - \{s, t\}$ doit avoir un excédent 0 ; en effet, d'après les lemmes 26.15 et 26.17 et d'après l'invariant selon lequel f est toujours un préflot, il n'y a pas de sommets débordants. Par conséquent, f est un flot. Comme h est une fonction de hauteur, le lemme 26.18 nous dit qu'il n'y a pas de chemin entre s et t dans le réseau résiduel G_f . D'après le théorème du flot maximum et de la coupe minimum, (théorème 26.7), f est donc un flot maximum. \square

e) Analyse de la méthode avec préflot

Pour montrer que l'algorithme de préflot générique se termine effectivement, nous allons borner le nombre d'opérations qu'il effectue. Chacun des trois types d'opération, à savoir ré-étiquetage, poussage saturant et poussage non saturant, est borné séparément. En connaissant ces bornes, il devient aisé de construire un algorithme qui s'exécute en $O(S^2A)$. Avant de plonger dans l'analyse, commençons toutefois par démontrer un lemme important.

Lemme 26.20 *Soit $G = (S, A)$ un réseau de transport de source s et de puits t et soit f un préflot de G . Alors, pour un sommet débordant quelconque u , il existe un chemin élémentaire de u vers s dans le réseau résiduel G_f .*

Démonstration : u étant un sommet débordant, posons $U = \{v : \text{il existe un chemin élémentaire de } u \text{ vers } v \text{ dans } G_f\}$ et supposons, en raisonnant par l'absurde, que $s \notin U$. Soit $\bar{U} = S - U$.

Nous affirmons que, pour chaque couple de sommets $v \in U$ et $w \in \bar{U}$, $f(w, v) \leq 0$. Pourquoi ? Si $f(w, v) > 0$, alors $f(v, w) < 0$, ce qui implique en retour que $c_f(v, w) = c(v, w) - f(v, w) > 0$. Il existe donc un arc $(v, w) \in A_f$, et donc il existe un chemin élémentaire de la forme $u \rightsquigarrow v \rightarrow w$ dans G_f , ce qui invalide notre choix de w .

On doit donc avoir $f(\bar{U}, U) \leq 0$, puisque chaque terme de cette sommation implicite est négatif ou nul.

D'où

$$\begin{aligned}
 e(U) &= f(S, U) && \text{(d'après l'équation (26.9))} \\
 &= f(\overline{U}, U) + f(U, U) && \text{(d'après le lemme 26.1, partie (3))} \\
 &= f(\overline{U}, U) && \text{(d'après le lemme 26.1, partie (1))} \\
 &\leq 0.
 \end{aligned}$$

Les excédents sont positifs ou nuls pour tous les sommets de $S - \{s\}$; comme nous avons fait l'hypothèse que $U \subseteq S - \{s\}$, on doit avoir $e(v) = 0$ pour tous les sommets $v \in U$. En particulier, $e(u) = 0$, ce qui contredit l'hypothèse selon laquelle u déborde. \square

Le lemme suivant borne les hauteurs des sommets et son corollaire borne le nombre d'opérations de ré-étiquetage qui sont effectuées au total.

Lemme 26.21 *Soit $G = (S, A)$ un réseau de transport de source s et de puits t . À chaque instant pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur G , on a $h[u] \leq 2|S| - 1$ pour tous les sommets $u \in S$.*

Démonstration : Les hauteurs de la source s et du puits t ne sont jamais modifiées, car ces sommets ne débordent jamais par définition. Donc, on a toujours $h[s] = |S|$ et $h[t] = 0$, valeurs qui sont toutes les deux pas plus grandes que $2|S| - 1$.

Considérons maintenant un sommet $u \in S - \{s, t\}$. Initialement, $h[u] = 0 \leq 2|S| - 1$. Nous allons montrer que, après chaque opération de ré-étiquetage, on a encore $h[u] \leq 2|S| - 1$. Quand u est réétiqueté, il est débordant, et le lemme 26.20 nous dit qu'il y a un chemin élémentaire p de u à s dans G_f . Soit $p = \langle v_0, v_1, \dots, v_k \rangle$, avec $v_0 = u$, $v_k = s$ et $k \leq |S| - 1$ car p est élémentaire. Pour $i = 0, 1, \dots, k - 1$, on a $(v_i, v_{i+1}) \in A_f$ et donc, d'après le lemme 26.17, $h[v_i] \leq h[v_{i+1}] + 1$. En développant ces inégalités sur le chemin p , on obtient $h[u] = h[v_0] \leq h[v_k] + k \leq h[s] + (|S| - 1) = 2|S| - 1$. \square

Corollaire 26.22 (Borne pour opérations de ré-étiquetage) *Soit $G = (S, A)$ un réseau de transport, de source s et de puits t . Alors, pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur G , le nombre d'opération de ré-étiquetage est au plus égal à $2|S| - 1$ par sommet et au plus égal à $(2|S| - 1)(|S| - 2) < 2|S|^2$ en tout.*

Démonstration : Seuls les sommets de $S - \{s, t\}$, au nombre de $|S| - 2$, peuvent être réétiquetés. Soit $u \in S - \{s, t\}$. L'opération RÉÉTIQUETER(u) augmente $h[u]$. Initialement, la valeur de $h[u]$ est 0 et, d'après le lemme 26.21, elle ne peut pas dépasser $2|S| - 1$. Donc, chaque sommet $u \in S - \{s, t\}$ est réétiqueté au plus $2|S| - 1$ fois, et le nombre total de ré-étiquetages effectués vaut au plus $(2|S| - 1)(|S| - 2) < 2|S|^2$. \square

Le lemme 26.21 aide aussi à borner le nombre de poussages saturants.

Lemme 26.23 (Borne pour poussages saturants) *Pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur un réseau de transport $G = (S, A)$ quelconque, le nombre de poussages saturants est inférieur à $2|S||A|$.*

Démonstration : Pour toute paire de sommets $u, v \in S$, on va compter ensemble les poussages saturants de u vers v et ceux de v vers u , en les appelant les poussages saturants entre u et v . S'il existe de tels poussages, l'un au moins des arcs (u, v) et (v, u) est en fait un arc de A . Maintenant, supposons qu'un poussage saturant de u vers v ait eu lieu. À cet instant, $h[v] = h[u] - 1$. Pour qu'un autre poussage de u vers v puisse avoir lieu ultérieurement, l'algorithme doit commencer par pousser du flot de v vers u , ce qui ne peut se produire que si $h[v] = h[u] + 1$. Comme $h[u]$ ne diminue jamais, pour que $h[v] = h[u] + 1$, il faut que la valeur de $h[v]$ augmente d'au moins 2. De même, $h[u]$ doit augmenter d'au moins 2 entre des poussages saturants de v vers u . Les hauteurs commencent à 0 et, d'après le lemme 26.21 elles ne dépassent jamais $2|S| - 1$, ce qui implique que le nombre de fois qu'un sommet voit sa hauteur augmenter de 2 est inférieur à $|S|$. Comme l'une au moins des valeurs $h[u]$ et $h[v]$ doit augmenter de 2 entre deux poussages saturants quelconques entre u et v , il y a moins de $2|S|$ poussages saturants entre u et v . En multipliant par le nombre d'arcs, on obtient une borne inférieure à $2|S||A|$ pour le nombre total de poussages saturants. \square

Le lemme suivant borne le nombre de poussages non saturants faits dans l'algorithme de préflot générique.

Lemme 26.24 (Borne pour poussages non saturants) *Pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur un réseau de transport $G = (S, A)$, le nombre de poussages non saturants est inférieur à $4|S|^2(|S| + |A|)$.*

Démonstration : Soit la fonction de potentiel $\Phi = \sum_{v:e(v)>0} h[v]$. Initialement $\Phi = 0$, et la valeur de Φ peut changer après chaque ré-étiquetage, poussage saturant ou poussage non saturant. On va borner la quantité par laquelle les poussages saturants et les ré-étiquetages peuvent contribuer à l'accroissement de Φ . On montrera ensuite que chaque poussage non saturant diminue forcément Φ d'au moins 1, et l'on utilisera ces bornes pour calculer un majorant du nombre de poussages non saturants.

Examinons les deux façons par lesquelles Φ pourrait augmenter. Primo, le ré-étiquetage d'un sommet u augmente Φ d'au moins $2|S|$, car l'ensemble sur lequel est calculée la somme est le même et le ré-étiquetage ne peut pas augmenter la hauteur de u de plus de sa hauteur maximale possible, laquelle, d'après le lemme 26.21, est au plus égale à $2|S| - 1$. Secundo, un poussage saturant d'un sommet u vers un sommet v augmente Φ de moins de $2|S|$, car aucune hauteur n'est modifiée et seul le sommet v , dont la hauteur est au plus $2|S| - 1$, pourrait éventuellement devenir débordant.

Montrons maintenant qu'un poussage non saturant de u vers v diminue Φ d'au moins 1. Pourquoi ? Avant le poussage, u était débordant ; quant à v , il pouvait l'être ou non. D'après le lemme 26.14, u n'est plus débordant après le poussage. De plus, v est forcément débordant après le poussage, sauf si c'est la source. Par conséquent, la fonction Φ a diminué de $h[u]$ exactement, et elle a augmenté soit de 0 soit de $h[v]$. Comme $h[u] - h[v] = 1$, l'effet net est que la fonction de potentiel a diminué d'au moins 1.

Donc, pendant l'exécution de l'algorithme, l'accroissement total de Φ est dû aux ré-étiquetages et aux poussages saturants ; et il est contraint, d'après le corollaire 26.22 et le lemme 26.23, d'être inférieur à $(2|S|)(2|S|^2) + (2|S|)(2|S||A|) = 4|S|^2(|S| + |A|)$. Comme $\Phi \geq 0$, la diminution totale, et donc le nombre total de poussages non saturants, est inférieure à $4|S|^2(|S| + |A|)$. \square

Ayant borné le nombre de ré-étiquetages, de poussages saturants et de poussages non saturants, nous avons tout ce qu'il faut pour analyser la procédure PRÉFLOT-GÉNÉRIQUE, et donc pour analyser tout algorithme fondé sur la méthode de préflot.

Théorème 26.25 *Pendant l'exécution de PRÉFLOT-GÉNÉRIQUE sur un réseau de transport $G = (S, A)$ quelconque, le nombre d'opérations élémentaires est $O(S^2A)$.*

Démonstration : Immédiate d'après le corollaire 26.22 et les lemmes 26.23 et 26.24. □

Ainsi, l'algorithme se termine après $O(S^2A)$ opérations. Tout ce qui reste à faire, c'est de donner une méthode efficace pour implémenter chaque opération et pour choisir une opération appropriée à exécuter.

Corollaire 26.26 *Il existe une implémentation de l'algorithme de préflot générique qui s'exécute en temps $O(S^2A)$ sur un réseau de transport $G = (S, A)$ quelconque.*

Démonstration : L'exercice 26.4.1 vous demandera de montrer comment implémenter l'algorithme générique avec une charge de $O(S)$ par opération de ré-étiquetage et de $O(1)$ par opération de poussage. Il vous demandera aussi de concevoir une structure de données qui permette de choisir une opération applicable en temps $O(1)$. Le corollaire s'en déduit. □

Exercices

26.4.1 Montrer comment implémenter l'algorithme de préflot générique avec un temps $O(S)$ par ré-étiquetage, un temps $O(1)$ par poussage et un temps $O(1)$ par sélection d'une opération applicable, soit un temps total de $O(S^2A)$.

26.4.2 Démontrer que l'algorithme de préflot générique dépense un temps total de $O(SA)$ seulement pour effectuer toutes les $O(S^2)$ opérations de ré-étiquetage

26.4.3 Supposons qu'un flot maximum ait été trouvé dans un réseau de transport $G = (S, A)$ à l'aide d'un algorithme de préflot. Donner un algorithme rapide pour trouver une coupe minimum dans G .

26.4.4 Donner un algorithme de préflot efficace permettant de trouver un couplage maximum dans un graphe biparti. Analyser votre algorithme.

26.4.5 Supposons que toutes les capacités d'arc d'un réseau de transport $G = (S, A)$ appartiennent à l'ensemble $\{1, 2, \dots, k\}$. Analyser le temps d'exécution de l'algorithme pousser-réétiqueter générique en fonction de $|S|$, $|A|$ et k . (*conseil* : Combien de fois chaque arc peut-il supporter un poussage non saturant avant de devenir saturé ?)

26.4.6 Montrer que la ligne 7 de INITIALISER-PRÉFLOT peut être modifiée ainsi

$$h[s] \leftarrow |S[G]| - 2$$

sans affecter la validité, ni les performances asymptotiques de l'algorithme de préflot générique.

26.4.7 Soit $\delta_f(u, v)$ la distance (nombre d'arcs) de u à v dans le réseau résiduel G_f . Montrer que PRÉFLOT-GÉNÉRIQUE conserve les propriétés suivantes : $h[u] < |S|$ implique que $h[u] \leq \delta_f(u, t)$, et $h[u] \geq |S|$ implique que $h[u] - |S| \leq \delta_f(u, s)$.

26.4.8 * Comme dans l'exercice précédent, soit $\delta_f(u, v)$ la distance de u à v dans le réseau résiduel G_f . Montrer comment l'algorithme de préflot générique peut être modifié pour conserver la propriété selon laquelle $h[u] < |S|$ implique que $h[u] = \delta_f(u, t)$ et que $h[u] \geq |S|$ implique que $h[u] - |S| = \delta_f(u, s)$. Le temps total que consacre votre implémentation à conserver cette propriété devra être $O(SA)$.

26.4.9 Montrer que le nombre de poussages non saturants exécutés par PRÉFLOT-GÉNÉRIQUE sur un réseau de transport $G = (S, A)$ vaut au plus $4|S|^2|A|$ pour $|S| \geq 4$.

26.5^{*} ALGORITHME RÉÉTIQUETER-VERS-L'AVANT

La méthode de préflot permet d'appliquer les opérations élémentaires dans n'importe quel ordre. Pourtant, en sélectionnant l'ordre soigneusement et en gérant la structure de données du réseau de manière efficace, on peut résoudre le problème du flot maximum plus rapidement qu'avec la borne $O(S^2A)$ donnée par le corollaire 26.26. Nous allons à présent étudier l'algorithme réétiqueter-vers-l'avant, qui est un algorithme de préflot dont le temps d'exécution est $O(S^3)$, ce qui est asymptotiquement au moins aussi bon que $O(S^2A)$, voire mieux pour les réseaux denses.

L'algorithme réétiqueter-vers-l'avant gère une liste des sommets du réseau. En commençant par le début, l'algorithme parcourt la liste, en choisissant de manière itérative un sommet débordant u puis en le « déchargeant », c'est-à-dire en effectuant des opérations de poussage et de ré-étiquetage jusqu'à ce que u n'ait plus d'excédent positif. Chaque fois qu'un sommet est réétiqueté, il est déplacé en début de liste (d'où le nom « réétiqueter-vers-l'avant ») et l'algorithme commence un nouveau parcours de la liste.

La validité et l'analyse de l'algorithme réétiqueter-vers-l'avant dépendent de la notion d'arcs « admissibles » : arcs du réseau résiduel à travers lesquels on peut pousser du flot. Après avoir démontré certaines propriétés concernant le réseau des arcs admissibles, nous étudierons l'opération de déchargement puis présenterons et analyserons l'algorithme réétiqueter-vers-l'avant lui-même.

a) *Arcs et réseaux admissibles*

Si $G = (S, A)$ est un réseau de transport, de source s et de puits t , f est un préflot de G et h est une fonction de hauteur, alors on dit que (u, v) est un **arc admissible** si $c_f(u, v) > 0$ et $h(u) = h(v) + 1$. Le **réseau admissible** est $G_{f,h} = (S, A_{f,h})$, où $A_{f,h}$ est l'ensemble des arcs admissibles.

Le réseau admissible est constitué des arcs à travers lesquels on peut pousser du flot. Le lemme suivant montre que ce réseau est un graphe orienté sans circuit.

Lemme 26.27 (Le réseau admissible est sans circuit) *Si $G = (S, A)$ est un réseau de transport, f est un préflot de G et h est une fonction de hauteur sur G , alors le réseau admissible $G_{f,h} = (S, A_{f,h})$ est sans circuit.*

Démonstration : La démonstration se fait par l'absurde. Supposons que $G_{f,h}$ contienne un circuit $p = \langle v_0, v_1, \dots, v_k \rangle$, où $v_0 = v_k$ et $k > 0$. Comme chaque arc de p est admissible, on a $h(v_{i-1}) = h(v_i) + 1$ pour $i = 1, 2, \dots, k$. En sommant le long du circuit, on obtient

$$\begin{aligned} \sum_{i=1}^k h(v_{i-1}) &= \sum_{i=1}^k (h(v_i) + 1) \\ &= \sum_{i=1}^k h(v_i) + k. \end{aligned}$$

Comme chaque sommet du circuit p apparaît une fois dans chacune des sommations, on arrive à la contradiction $0 = k$. \square

Les deux lemmes suivants montrent comment les opérations de poussage et de ré-étiquetage modifient le réseau admissible.

Lemme 26.28 *Soit $G = (S, A)$ un réseau de transport, soit f un préflot de G , et supposons que l'attribut h soit une fonction de hauteur. Si un sommet u déborde et si (u, v) est un arc admissible, alors **POUSSER** (u, v) peut s'appliquer. L'opération ne crée aucun arc admissible nouveau, mais elle peut entraîner que (u, v) ne soit plus admissible.*

Démonstration : D'après la définition d'un arc admissible, du flot peut être poussé de u vers v . Comme u déborde, l'opération **POUSSER** (u, v) s'applique. Le seul arc résiduel nouveau qui peut être créé en poussant du flot de u vers v est l'arc (v, u) . Comme $h(v) = h(u) - 1$, l'arc (v, u) ne peut pas devenir admissible. Si l'opération est un poussage saturant, alors $c_f(u, v) = 0$ après l'opération et (u, v) devient inadmissible. \square

Lemme 26.29 *Soit $G = (S, A)$ un réseau de transport, soit f un préflot de G , et supposons que l'attribut h soit une fonction de hauteur. Si un sommet u déborde et qu'il n'existe aucun arc admissible partant de u , alors **RÉÉTIQUETER** (u) s'applique. Après l'opération de ré-étiquetage, il existe au moins un arc admissible partant de u mais il n'existe aucun arc arrivant sur u .*

Démonstration : Si u déborde, alors d'après le lemme 26.15, on peut lui appliquer soit un poussage, soit un ré-étiquetage. S'il n'existe aucun arc admissible quittant u , alors aucun flot ne peut être poussé à partir de u et donc RÉÉTIQUETER(u) s'applique. Après l'opération de ré-étiquetage, $h[u] = 1 + \min \{h[v] : (u, v) \in A_f\}$. Donc, si v est un sommet pour qui le minimum est atteint dans cet ensemble, l'arc (u, v) devient admissible. Donc, après le ré-étiquetage, il existe au moins un arc admissible sortant de u .

Pour montrer qu'aucun arc admissible n'entre dans u après un ré-étiquetage, supposons qu'il existe un sommet v tel que (v, u) soit admissible. Alors, $h[v] = h[u] + 1$ après le ré-étiquetage et donc $h[v] > h[u] + 1$ juste avant. Mais d'après le lemme 26.13, aucun arc résiduel n'existe entre des sommets dont les hauteurs diffèrent de plus de 1. Par ailleurs, le ré-étiquetage d'un sommet ne modifie pas le réseau résiduel. Donc, (v, u) n'appartient pas au réseau résiduel et, partant, ne peut pas appartenir au réseau admissible. \square

b) Listes de voisinage

Dans l'algorithme réétiqueter-vers-l'avant, les arcs sont organisés en « listes de voisinage ». Étant donné un réseau de transport $G = (S, A)$, la **liste de voisinage** $N[u]$ d'un sommet $u \in S$ est une liste simplement chaînée des voisins de u dans G . Donc, le sommet v apparaît dans la liste $N[u]$ si $(u, v) \in A$ ou $(v, u) \in A$. La liste de voisinage $N[u]$ contient exactement les sommets v pour lesquels il peut exister un arc résiduel (u, v) . Le premier sommet de $N[u]$ est pointé par $tête[N[u]]$. Le sommet qui suit v dans une liste de voisinage est pointé par $voisin-suivant[v]$; ce pointeur vaut NIL si v est le dernier sommet de la liste de voisinage.

L'algorithme réétiqueter-vers-l'avant parcourt chaque liste de voisinage dans un ordre arbitraire, fixé pour toute la durée d'exécution de l'algorithme. Pour chaque sommet u , le champ $courant[u]$ pointe vers le sommet en cours de traitement dans $N[u]$. Initialement, $courant[u]$ a la valeur $tête[N[u]]$.

c) Déchargement d'un sommet débordant

Un sommet débordant u est **déchargé** via poussage de tout son flot excédentaire à travers les arcs admissibles conduisant aux sommets voisins, avec ré-étiquetage de u si besoin est pour rendre admissibles les arcs quittant u . Ces étapes sont traduites par le pseudo code suivant.

```

DÉCHARGER( $u$ )
1  tant que  $e[u] > 0$ 
2    faire  $v \leftarrow courant[u]$ 
3    si  $v = \text{NIL}$ 
4      alors RÉÉTIQUETER( $u$ )
5       $courant[u] \leftarrow tête[N[u]]$ 
6    sinon si  $c_f(u, v) > 0$  et  $h[u] = h[v] + 1$ 
7      alors POUSSER( $u, v$ )
8    sinon  $courant[u] \leftarrow voisin-suivant[v]$ 

```

La figure 26.9 illustre plusieurs itérations de la boucle **tant que** des lignes 1–8, qui s'exécute aussi longtemps que le sommet u a un excédent positif. Chaque itération effectue exactement une action parmi trois possibles, en fonction du sommet courant v dans la liste de voisinage $N[u]$.

- 1) Si v vaut NIL, c'est que la fin de $N[u]$ est atteinte. La ligne 4 ré-étiquète le sommet u et la ligne 5 réinitialise le voisin courant de u pour qu'il soit le premier dans $N[u]$. (Le lemme 26.30 ci-après dit que l'opération de ré-étiquetage s'applique dans cette situation.)
- 2) Si v ne vaut pas NIL et que (u, v) est un arc admissible (déterminé par le test de la ligne 6), alors la ligne 7 pousse une partie (voire la totalité) du flot excédentaire de u vers le sommet v .
- 3) Si v ne vaut pas NIL mais que (u, v) n'est pas admissible, alors la ligne 8 *avance* $\text{courant}[u]$ d'une position dans la liste de voisinage $N[u]$.

Observez que, si DÉCHARGER est appelé sur un sommet débordant u , alors la dernière action effectuée par DÉCHARGER doit être un poussage depuis u . Pourquoi ? La procédure ne se termine que quand $e[u]$ devient égal à zéro, et ni l'opération de ré-étiquetage ni l'avancement du pointeur $\text{courant}[u]$ n'affecte la valeur de $e[u]$.

Il faut être certain que, quand POUSSER ou RÉÉTIQUETER est appelée par DÉCHARGER, l'opération peut s'appliquer. Le lemme suivant établit ce fait.

Lemme 26.30 *Si DÉCHARGER appelle POUSSER(u, v) à la ligne 7, c'est qu'une opération de poussage est applicable sur (u, v) . Si DÉCHARGER appelle RÉÉTIQUETER(u) à la ligne 4, c'est qu'une opération de ré-étiquetage s'applique à u .*

Démonstration : Les tests des lignes 1 et 6 garantissent qu'un poussage n'a lieu que s'il peut s'appliquer, ce qui prouve la première affirmation du lemme.

Pour démontrer la deuxième affirmation, il suffit, d'après le test de la ligne 1 et le lemme 26.29, de montrer qu'aucun arc sortant de u n'est admissible. Observez que, au cours des appels répétés à DÉCHARGER(u), le pointeur $\text{courant}[u]$ parcourt séquentiellement la liste $N[u]$. Chaque « passe » commence en tête de $N[u]$ et se termine avec $\text{courant}[u] = \text{NIL}$, stade où u est réétiqueté et où une nouvelle passe commence. Pour que le pointeur $\text{courant}[u]$ dépasse un sommet $v \in N[u]$ lors d'une passe, l'arc (u, v) doit être déclaré non admissible par le test de la ligne 6. Donc, quand la passe se termine, tout arc quittant u a été déclaré non admissible à un certain moment de la passe. L'observation clé est que, à la fin de la passe, les arcs sortant de u ne sont toujours pas admissibles. Pourquoi ? D'après le lemme 26.28, les poussages ne peuvent pas créer d'arc admissible, et encore moins un arc partant de u . Donc, un arc admissible est forcément créé par une opération de ré-étiquetage. Mais le sommet u n'est pas réétiqueté pendant la passe et, d'après le lemme 26.29, tout autre sommet v qui est réétiqueté pendant la passe ne possède aucun arc entrant admissible après le ré-étiquetage. Ainsi, à la fin de la passe, tous les arcs sortant de u restent inadmissibles, ce qui démontre le lemme. \square

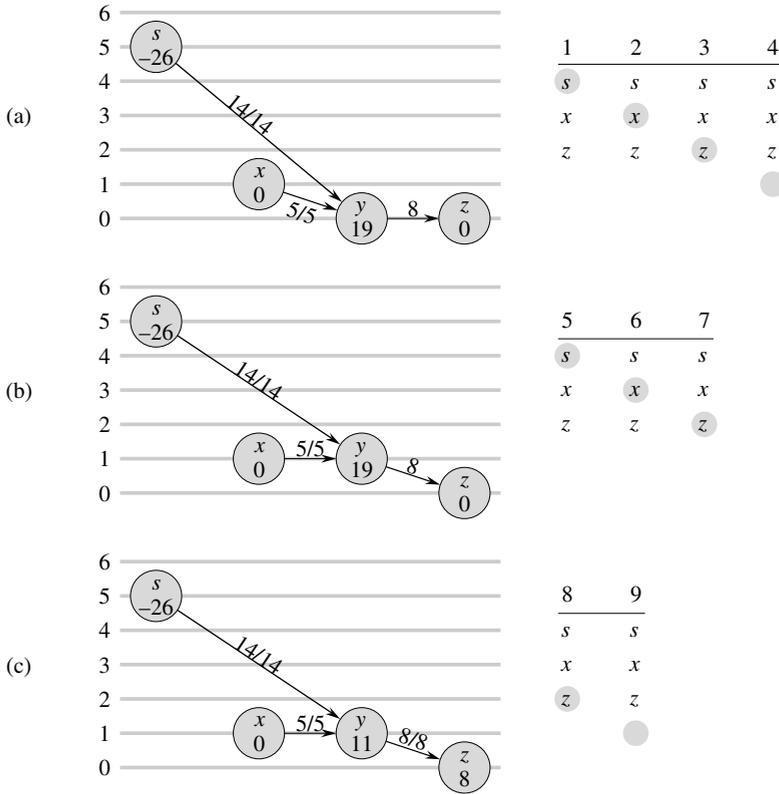
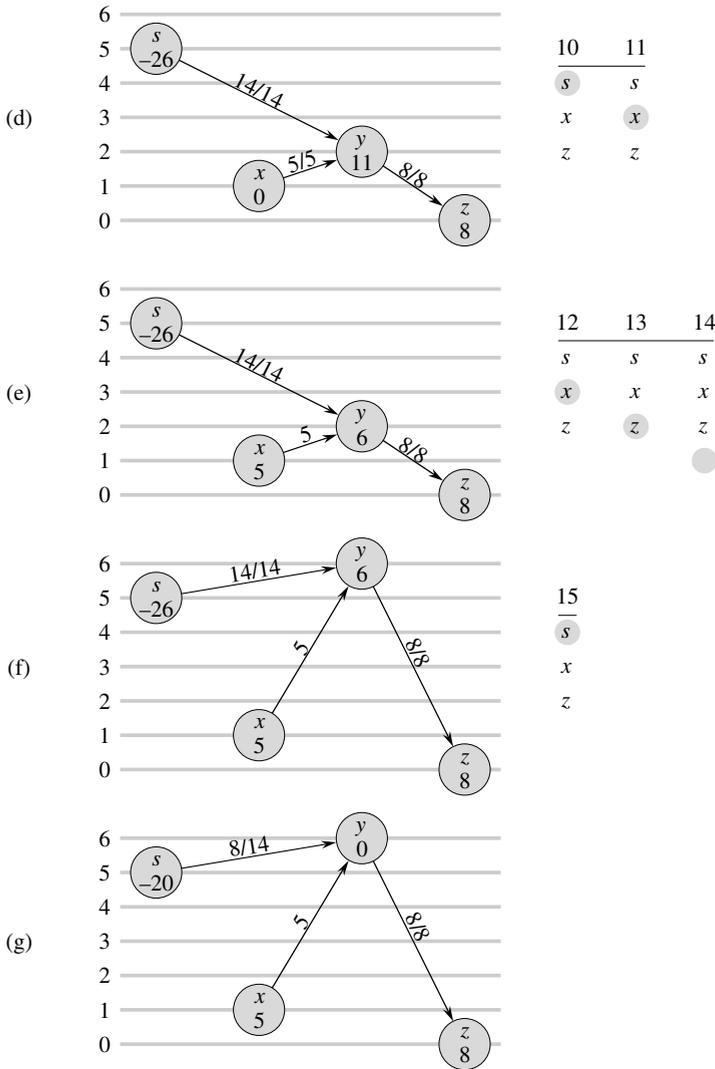


Figure 26.9 Déchargement d'un sommet. La boucle **tant que** de DÉCHARGER est itérée 15 fois pour pousser tout le flot excédentaire du sommet y . Seuls les voisins de y et les arcs arrivant à ou partant de y sont représentés. Dans chaque partie, le nombre apparaissant à l'intérieur de chaque sommet représente son excédent au début de la première itération montrée dans la partie, et chaque sommet est représenté à la hauteur courante. La liste de voisinage $N[y]$ au début de chaque itération apparaît à droite, avec au-dessus le compteur d'itérations. Le voisin en gris est $current[y]$. (a) Initialement, il faut pousser 19 unités de flot excédentaires à pousser depuis y et $current[y] = s$. Les itérations 1, 2 et 3 se contentent d'avancer $current[y]$, puisqu'il n'existe aucun arc admissible sortant de y . À l'itération 4, $current[y] = NIL$ (représenté par la pastille en gris située sous la liste de voisinage), et donc y est réétiqueté et $current[y]$ est réinitialisé pour désigner le début de la liste. (b) Après ré-étiquetage, le sommet y a une hauteur égale à 1. Aux itérations 5 et 6, les arcs (y, s) et (y, x) ne sont pas admissibles mais 8 unités de flot excédentaire sont poussées de y vers z à l'itération 7. À cause du poussage, $current[y]$ n'avance pas dans cette itération. (c) Le poussage de l'itération 7 ayant saturé l'arc (y, z) , il n'est plus admissible à l'itération 8. À l'itération 9, $current[y] = NIL$, et donc le sommet y est réétiqueté à nouveau et $current[y]$ est réinitialisé. (d) À l'itération 10, (y, s) n'est pas admissible mais 5 unités de flot excédentaire sont poussées de y vers x à l'itération 11. (e) Comme $current[y]$ n'a pas avancé pendant l'itération 11, (y, x) n'est pas admissible lors de l'itération 12. L'itération 13 trouve (y, z) inadmissible, et l'itération 14 ré-étiquète le sommet y et réinitialise $current[y]$. (f) L'itération 15 pousse 6 unités de flot excédentaire de y vers s . (g) Le sommet y n'a plus de flot excédentaire et DÉCHARGER se termine. Dans cet exemple, DÉCHARGER commence et finit avec le pointeur courant en tête de liste, ce qui n'est pas forcément le cas en général.



d) Algorithme réétiqueter-vers-l'avant

Dans l'algorithme réétiqueter-vers-l'avant, on gère une liste chaînée L composée de tous les sommets de $S - \{s, t\}$. Une propriété essentielle est que les sommets de L sont triés topologiquement selon le réseau admissible, comme nous le verrons dans l'invariant de boucle donné plus loin. (Rappelez-vous que, d'après le lemme 26.27, le réseau admissible est un graphe orienté sans circuit.)

Le pseudo code de l'algorithme réétiqueter-vers-l'avant suppose que les listes de voisinage $N[u]$ ont déjà été créées pour chaque sommet u . Il fait également l'hypothèse que $suivant[u]$ pointe vers le sommet qui suit u dans la liste L et que, comme d'habitude, $suivant[u] = \text{NIL}$ si u est le dernier sommet de la liste.

```

RÉÉTIQUETER-VERS-L'AVANT( $G, s, t$ )
1  INITIALISER-PRÉFLOT( $G, s$ )
2   $L \leftarrow S[G] - \{s, t\}$ , dans un ordre quelconque
3  pour chaque sommet  $u \in S[G] - \{s, t\}$ 
4      faire  $courant[u] \leftarrow tête[N[u]]$ 
5   $u \leftarrow tête[L]$ 
6  tant que  $u \neq \text{NIL}$ 
7      faire  $ancienne-hauteur \leftarrow h[u]$ 
8          DÉCHARGER( $u$ )
9          si  $h[u] > ancienne-hauteur$ 
10             alors déplacer  $u$  vers le début de la liste  $L$ 
11              $u \leftarrow suivant[u]$ 

```

L'algorithme réétiqueter-vers-l'avant fonctionne de la manière suivante. La ligne 1 initialise le préflot et les hauteurs aux mêmes valeurs que celles de l'algorithme de préflot générique. La ligne 2 initialise la liste L pour qu'elle contienne tous les sommets potentiellement débordants, dans un ordre quelconque. Les lignes 3–4 initialisent le pointeur *courant* de chaque sommet u pour le faire pointer vers le premier sommet de la liste de voisinage de u .

Comme le montre la figure 26.10, la boucle **tant que** des lignes 6–11 parcourt la liste L , en déchargeant les sommets. La ligne 5 la fait débiter avec le premier sommet de la liste. À chaque nouvelle itération, un sommet u est déchargé à la ligne 8. Si u a été réétiqueté par la procédure DÉCHARGER, la ligne 10 le déplace vers la tête de la liste L . Cette décision est faite via mémorisation de la hauteur de u dans la variable *ancienne-hauteur* avant déchargement (ligne 7), puis comparaison avec la hauteur de u après déchargement (ligne 9). La ligne 11 permet à l'itération suivante de la boucle **tant que** d'utiliser le sommet qui suit u dans la liste L . Si u a été déplacé vers la tête de la liste, le sommet utilisé dans l'itération suivante est celui qui suit u dans sa nouvelle position dans la liste.

Pour montrer que RÉÉTIQUETER-VERS-L'AVANT calcule un flot maximum, nous allons montrer que c'est une implémentation de l'algorithme de préflot générique. Commençons par remarquer qu'il n'effectue d'opérations de poussage et de ré-étiquetage que lorsqu'elles sont applicables, puisque le lemme 26.30 garantit que DÉCHARGER ne les exécute que si elles sont applicables. Il reste à montrer que quand RÉÉTIQUETER-VERS-L'AVANT se termine, il n'y a plus d'opération élémentaire applicable. Le reste de la démonstration de validité repose sur l'invariant de boucle que voici :

À chaque test fait en ligne 6 de RÉÉTIQUETER-VERS-L'AVANT, la liste L est un tri topologique des sommets du réseau admissible $G_{f,h} = (S, A_{f,h})$, et aucun sommet avant u dans la liste n'a de flot excédentaire.

Initialisation : Juste après que INITIALISER-PRÉFLOT a été exécuté, $h[s] = |S|$ et $h[v] = 0$ pour tout $v \in S - \{s\}$. Comme $|S| \geq 2$ (car S contient au moins s et t), aucun arc ne peut être admissible. Donc, $A_{f,h} = \emptyset$ et tout tri de $S - \{s, t\}$ est un ordre topologique de $G_{f,h}$.

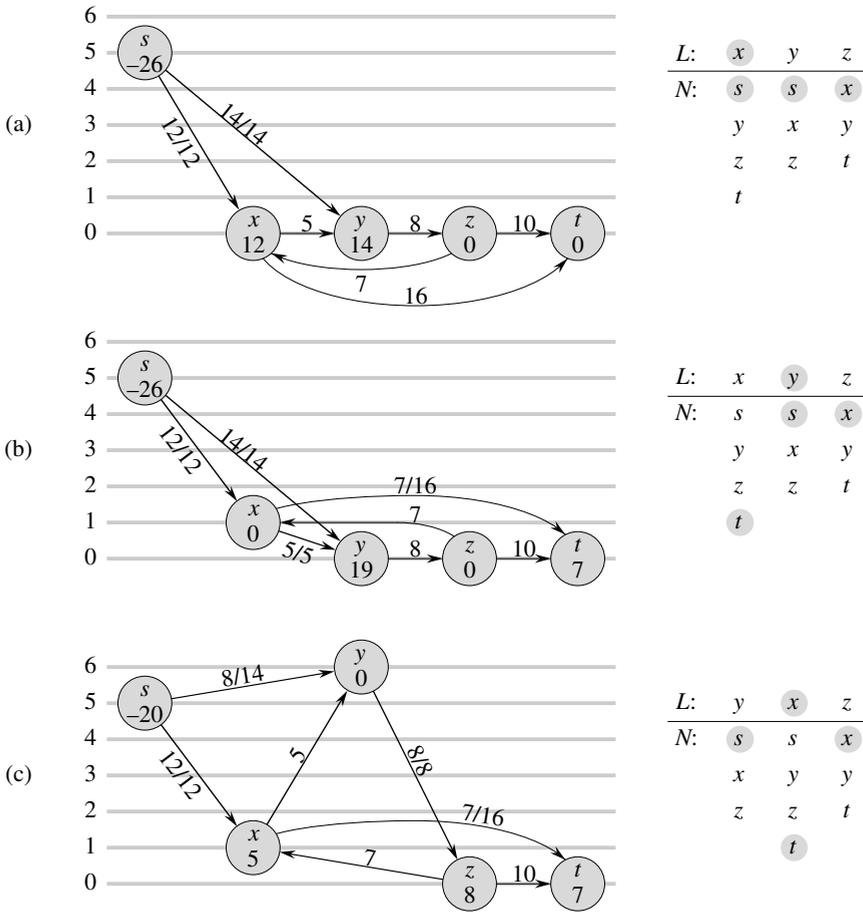
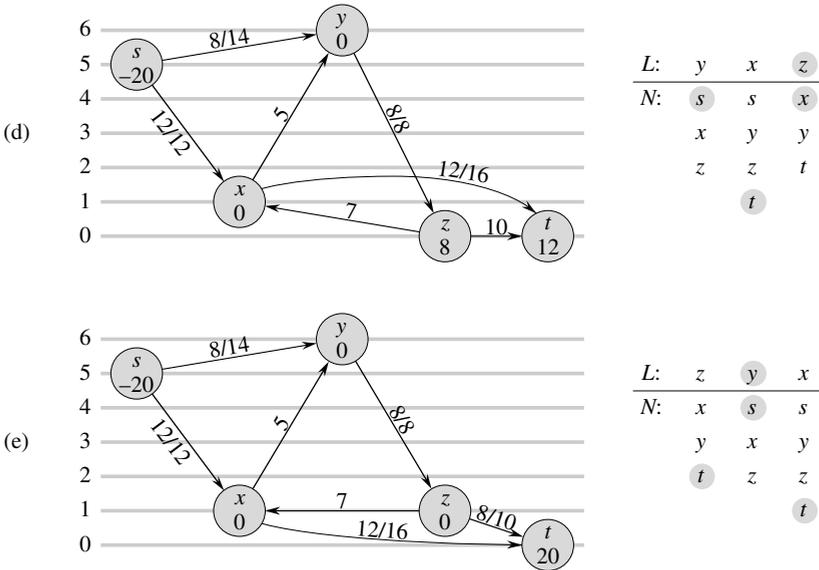


Figure 26.10 L'action de RÉÉTIQUETER-VERS-L' AVANT. (a) Un réseau de flot juste avant la première itération de la boucle tant que. Au départ, 26 unités de flot quittent la source s . On voit sur la droite la liste $L = (x, y, z)$ initiale, où $u = x$. Sous chaque sommet de la liste L se trouve sa liste de voisinage, avec en gris son voisin courant. Le sommet x est déchargé. Il est réétiqueté avec la hauteur 1, 5 unités de flot excédentaires sont poussées vers y et les 7 unités excédentaires restantes sont poussées vers le puits t . Puisque x est réétiqueté, il est déplacé en tête de L , ce qui dans ce cas ne modifie pas la structure de L . (b) Après x , le sommet suivant de L est y . La figure 26.9 montre l'action détaillée du déchargement de y dans cette situation. Comme y est réétiqueté, il est déplacé vers la tête de L . (c) Le sommet x suit maintenant y dans L et il est déchargé à nouveau, ce qui pousse les 5 unités de flot excédentaires vers t . Comme le sommet x n'est pas réétiqueté pendant cette opération de déchargement, il reste à sa place dans la liste L . (d) Comme le sommet z suit le sommet x dans L , il est déchargé. Il est réétiqueté avec la hauteur 1 et les 8 unités de flot excédentaires sont poussées vers t . Puisque z est réétiqueté, il est déplacé vers le début de la liste L . (e) Le sommet y suit maintenant le sommet z dans L et est donc déchargé. Mais comme y n'a plus d'excédent, DÉCHARGER rend la main immédiatement et y reste à sa place dans L . Le sommet x est alors déchargé. Puisque lui non plus n'a plus d'excédent, DÉCHARGER rend encore la main et x reste à sa place dans L . RÉÉTIQUETER-VERS-L' AVANT a atteint la fin de la liste L et se termine. Il n'existe plus de sommets débordants et le préflot est un flot maximum.



Comme u est initialement la tête de la liste L , il n'y a pas de sommet avant lui, et par conséquent il n'y a pas de sommet excédentaire avant lui.

Conservation : Pour voir que l'ordre topologique est conservé par chaque itération de la boucle **tant que**, commençons par observer que le réseau admissible n'est modifié que par des opérations de poussage et de ré-étiquetage. D'après le lemme 26.28, les poussages n'entraînent pas que des arcs deviennent admissibles. Donc, les arcs admissibles ne peuvent être créés que par des ré-étiquetages. Après qu'un sommet u a été réétiqueté, cependant, le lemme 26.29 dit qu'il n'y a pas d'arc admissible qui arrive dans u mais qu'il peut y avoir des arcs admissibles qui partent de u . Donc, en déplaçant u vers le début de L , l'algorithme garantit qu'aucun arc admissible partant de u ne vérifie l'ordre du tri topologique.

Pour voir qu'aucun sommet précédant u dans L n'a de flot excédentaire, notons u' le sommet qui sera u dans itération suivante. Les sommets qui précéderont u' dans itération suivante incluent le u courant (en vertu de la ligne 11), plus soit aucun autre sommet (si u est réétiqueté) soit les mêmes sommets qu'auparavant (si u n'est pas réétiqueté). Comme u est déchargé, il n'a pas de flot excédentaire après. Donc, si u est réétiqueté pendant le déchargement, aucun sommet précédant u' n'a de flot excédentaire. Si u n'est pas réétiqueté lors du déchargement, aucun sommet situé avant lui dans la liste n'a acquis de flot excédentaire pendant ce déchargement, car L est restée en permanence triée topologiquement pendant le déchargement (comme précédemment mentionné, il n'y a création d'arcs admissibles que via ré-étiquetage, pas via poussage); donc chaque opération de poussage force le flot excédentaire à n'aller que vers les sommets situés plus loin dans la liste (ou vers s ou t). À nouveau, aucun sommet précédant u' n'a de flot excédentaire.

Terminaison : Quand la boucle se termine, u vient de dépasser la fin de L , et donc l'invariant de boucle garantit que l'excédent de chaque sommet est 0. Par conséquent, aucune opération élémentaire n'est applicable.

e) *Analyse*

Nous allons maintenant montrer que RÉÉTIQUETER-VERS-L'AVANT s'exécute en temps $O(S^3)$ sur un réseau de transport $G = (S, A)$ quelconque. Puisque l'algorithme est une implémentation de l'algorithme de préflot générique, on tire parti du corollaire 26.22, qui fournit une borne $O(S)$ sur le nombre d'opérations de ré-étiquetage exécutées par sommet et une borne $O(S^2)$ sur le nombre total de ré-étiquetages. De plus, l'exercice 26.4.2 donne une borne $O(SA)$ pour le temps total passé à exécuter des opérations de ré-étiquetage et le lemme 26.23 donne une borne $O(SA)$ sur le nombre total de poussages saturants.

Théorème 26.31 *Le temps d'exécution de RÉÉTIQUETER-VERS-L'AVANT sur un réseau de transport $G = (S, A)$ quelconque est $O(S^3)$.*

Démonstration : Appelons « phase » de l'algorithme réétiqueter-vers-l'avant l'intervalle de temps entre deux ré-étiquetages consécutifs. Il y a $O(S^2)$ phases, puisqu'il y a $O(S^2)$ ré-étiquetages. Chaque phase est composée d'au plus $|S|$ appels à DÉCHARGER, ce qu'on peut justifier comme suit. Si DÉCHARGER n'effectue aucun ré-étiquetage, le prochain appel à DÉCHARGER se fait plus avant dans la liste L et la longueur de L est inférieure à $|S|$. Si DÉCHARGER effectue un ré-étiquetage, le prochain appel à DÉCHARGER appartient à une phase différente. Comme chaque phase contient au plus $|S|$ appels à DÉCHARGER et qu'il existe $O(S^2)$ phases, le nombre d'appels à DÉCHARGER en ligne 8 de RÉÉTIQUETER-VERS-L'AVANT est $O(S^3)$. Donc, le travail total effectué par la boucle **tant que** de RÉÉTIQUETER-VERS-L'AVANT, sans compter le travail effectué par DÉCHARGER, est au plus $O(S^3)$.

Il faut ensuite borner le travail effectué par DÉCHARGER pendant l'exécution de l'algorithme. Chaque itération de la boucle **tant que** à l'intérieur de DÉCHARGER exécute une action parmi trois possibles. Nous devons analyser la quantité totale de travail mise en œuvre pendant la réalisation de chacune de ces actions.

Commençons par les ré-étiquetages (lignes 4–5). L'exercice 26.4.2 donne une borne $O(SA)$ pour l'exécution des $O(S^2)$ ré-étiquetages.

Maintenant supposons que l'action met à jour le pointeur *courant*[u] en ligne 8. Cette action intervient $O(\text{degré}(u))$ fois pour chaque ré-étiquetage d'un sommet u et $O(S \cdot \text{degré}(u))$ fois au total pour le sommet. La quantité totale de travail effectuée pour tous les sommets pendant la progression des pointeurs dans les listes de voisinage est donc $O(SA)$ d'après le lemme de la poignée de main (exercice B.4.1).

Le troisième type d'action effectué par DÉCHARGER est une opération de poussage (ligne 7). Nous savons déjà que le nombre total de poussages saturants est $O(SA)$. Notez que, si un poussage non saturant est exécuté, DÉCHARGER rend la main immédiatement, puisque le poussage ramène l'excédent à 0. Il peut donc y avoir au plus un poussage non saturant par appel à DÉCHARGER. Comme on l'a vu, DÉCHARGER est appelée $O(S^3)$ fois ; et donc le temps total dépensé pour effectuer des poussages non saturants est $O(S^3)$.

Le temps d'exécution de RÉÉTIQUETER-VERS-L'AVANT est donc $O(S^3 + SA)$, soit $O(S^3)$. □

Exercices

26.5.1 Illustrer l'exécution de RÉÉTIQUETER-VERS-L'AVANT à la manière de la figure 26.10 pour le réseau de transport de la figure 26.1(a). On suppose que l'ordre initial des sommets dans L est $\langle v_1, v_2, v_3, v_4 \rangle$ et que les listes de voisinage sont

$$\begin{aligned} N[v_1] &= \langle s, v_2, v_3 \rangle, \\ N[v_2] &= \langle s, v_1, v_3, v_4 \rangle, \\ N[v_3] &= \langle v_1, v_2, v_4, t \rangle, \\ N[v_4] &= \langle v_2, v_3, t \rangle. \end{aligned}$$

26.5.2 * On voudrait implémenter un algorithme de préflot pour lequel on gère une file de sommets débordants. L'algorithme décharge itérativement le sommet en tête de file, et tous les sommets qui ne débordaient pas avant le déchargement mais qui débordent après sont placés à la fin de la file. Après déchargement du sommet situé en tête, celui-ci est supprimé. Quand la file est vide, l'algorithme se termine. Montrer que cet algorithme peut être implémenté pour calculer un flot maximum en temps $O(S^3)$.

26.5.3 Montrer que l'algorithme générique fonctionne encore si RÉÉTIQUETER met à jour $h[u]$ en se contentant de calculer $h[u] \leftarrow h[u] + 1$. En quoi cela affecte-t-il l'analyse de RÉÉTIQUETER-VERS-L'AVANT ?

26.5.4 * Montrer que si l'on décharge systématiquement un sommet débordant de hauteur maximale, on peut faire en sorte que la méthode de préflot s'exécute en temps $O(S^3)$.

26.5.5 Supposons qu'à un certain point de l'exécution d'un algorithme de préflot, il existe un entier $0 < k \leq |S| - 1$ pour lequel aucun sommet ne vérifie $h[v] = k$. Montrer que tous les sommets ayant $h[v] > k$ sont sur le côté source d'une coupe minimum. S'il existe un tel k , l'*heuristique du fossé* actualise chaque sommet $v \in S - s$ pour lequel $h[v] > k$ de façon à faire l'affectation $h[v] \leftarrow \max(h[v], |S| + 1)$. Montrer que l'attribut résultant h est une fonction de hauteur. (L'heuristique du fossé est vitale pour assurer les bonnes performances concrètes des implémentations de la méthode de préflot.)

PROBLÈMES

26.1. Le problème de l'évacuation

Une *grille* $n \times n$ est un graphe non orienté constitué de n lignes et n colonnes de sommets, comme illustré sur la figure 26.11. Le sommet situé sur la i ème ligne et la j ème colonne est noté (i, j) . Tous les sommets de la grille ont exactement quatre voisins, sauf les sommets situés au bord, qui sont les points (i, j) pour lesquels $i = 1$, $i = n$, $j = 1$ ou $j = n$.

Étant donnés $m \leq n^2$ points de départ $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ dans la grille, le **problème de l'évacuation** consiste à déterminer s'il existe m chemins sans sommet commun reliant les points de départ à m points différents quelconques situés au bord de la grille. Par exemple, la grille de la figure 26.11(a) possède une évacuation, mais celle de la figure 26.11(b) n'en possède pas.

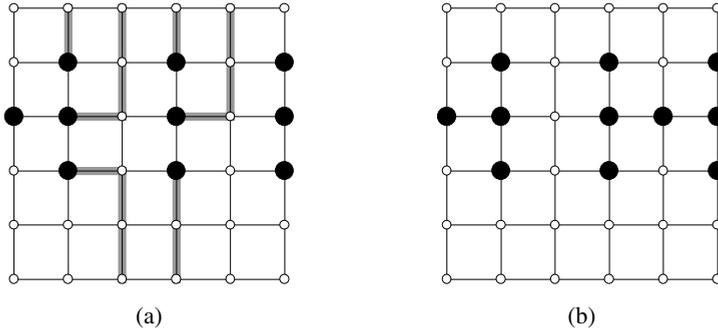


Figure 26.11 Grilles pour le problème de l'évacuation. Les points de départ sont en noir et les autres sommets de la grille sont en blanc. (a) Une grille avec une évacuation, représentée par des chemins ombrés. (b) Une grille sans évacuation.

- On considère un réseau de transport dans lequel les sommets, ainsi que les arcs, possèdent des capacités. Autrement dit, le flot positif total entrant dans un sommet donné est soumis à une contrainte de capacité. Montrer que, dans un réseau où arcs et sommets ont des capacités, la détermination du flot maximum peut se ramener à un problème de flot maximum ordinaire sur un réseau de transport de taille comparable.
- Décrire un algorithme efficace permettant de résoudre le problème de l'évacuation et analyser son temps d'exécution.

26.2. Couverture de chemins minimum

Une **couverture de chemins** d'un graphe orienté $G = (S, A)$ est un ensemble P de chemins sans sommet commun, tels que tout sommet de S soit inclus dans exactement un chemin de P . Les chemins peuvent commencer et se terminer n'importe où et être d'une longueur quelconque, y compris nulle. Une **couverture de chemins minimum** de G est une couverture de chemins contenant le moins de chemins possible.

- Donner un algorithme efficace permettant de trouver une couverture de chemins minimum d'un graphe orienté sans circuit $G = (S, A)$. (*Conseil* : En supposant que $S = \{1, 2, \dots, n\}$, construire le graphe $G' = (S', A')$, où

$$S' = \{x_0, x_1, \dots, x_n\} \cup \{y_0, y_1, \dots, y_n\} ,$$

$$A' = \{(x_0, x_i) : i \in S\} \cup \{(y_i, y_0) : i \in S\} \cup \{(x_i, y_j) : (i, j) \in A\} ,$$

et exécuter un algorithme de flot maximum.)

- b. Votre algorithme fonctionne-t-il pour les graphes orientés contenant des circuits ?
Dire pourquoi.

26.3. Expérimentations de navette spatiale

Le professeur Spock est consultant pour la NASA, qui projette une série de vols de navette spatiale et doit décider quelles sont les expériences commerciales à effectuer et quels sont les instruments à emporter à bord pour chaque vol. Pour chaque vol, la NASA considère un ensemble $E = \{E_1, E_2, \dots, E_m\}$ d'expériences et le sponsor de l'expérience E_j a accepté de payer p_j dollars à la NASA pour les résultats de l'expérience. Les expériences nécessitent un ensemble $I = \{I_1, I_2, \dots, I_n\}$ d'instruments ; chaque expérience E_j nécessite tous les instruments d'un sous-ensemble $R_j \subseteq I$. Le coût du transport de l'instrument I_k est c_k dollars. Le travail du professeur consiste à trouver un algorithme efficace pour déterminer quelles sont les expériences à effectuer et quels sont les instruments à emporter pour un vol donné, de manière à maximiser le revenu net, qui est la différence entre les rentrées dues aux expériences effectuées et le coût total des instruments transportés.

On considère le réseau G suivant. Il contient un sommet source s , les sommets I_1, I_2, \dots, I_n , les sommets E_1, E_2, \dots, E_m et un sommet puits t . Pour $k = 1, 2, \dots, n$, il existe un arc (s, I_k) de capacité c_k et pour $j = 1, 2, \dots, m$, il existe un arc (E_j, t) de capacité p_j . Pour $k = 1, 2, \dots, n$ et $j = 1, 2, \dots, m$, si $I_k \in R_j$, alors il existe un arc (I_k, E_j) de capacité infinie.

- Montrer que, si $E_j \in T$ pour une coupe (S, T) de G de capacité finie, alors $I_k \in T$ pour chaque $I_k \in R_j$.
- Montrer comment déterminer le revenu net maximal à partir de la capacité de la coupe minimum de G et des valeurs p_j données.
- Donner un algorithme efficace permettant de déterminer quelles sont les expériences à effectuer et les instruments à emporter. Analyser le temps d'exécution de votre algorithme en fonction de m, n et $r = \sum_{j=1}^m |R_j|$.

26.4. Mise à jour du flot maximum

Soit $G = (S, A)$ un réseau de transport, de source s et de puits t , à capacités entières. Supposons qu'on connaisse un flot maximum de G .

- Supposons que la capacité d'un arc individuel $(u, v) \in A$ soit augmentée de 1. Donner un algorithme à temps $O(S + A)$ permettant de mettre à jour le flot maximum.
- Supposons que la capacité d'un arc individuel $(u, v) \in A$ soit diminuée de 1. Donner un algorithme à temps $O(S + A)$ permettant de mettre à jour le flot maximum.

26.5. Flot maximum via échelonnement

Soit $G = (S, A)$ un réseau de transport, de source s et de puits t , ayant une capacité entière $c(u, v)$ sur chaque arc $(u, v) \in A$. Soit $C = \max_{(u,v) \in A} c(u, v)$.

- a. Montrer qu'une coupe minimum de G a une capacité au plus égale à $C|A|$.
- b. Pour un nombre K donné, montrer qu'un chemin améliorant de capacité au moins égale à K peut être trouvé en temps $O(A)$, si un tel chemin existe.

On modifie MÉTHODE-FORD-FULKERSON de la façon suivante, pour calculer un flot maximum de G .

FLOT-MAX-ET-ECHELONNEMENT(G, s, t)

```

1   $C \leftarrow \max_{(u,v) \in A} c(u, v)$ 
2  initialiser flot  $f$  à 0
3   $K \leftarrow 2^{\lceil \lg C \rceil}$ 
4  tant que  $K \geq 1$ 
5      faire tant que il existe un chemin améliorant  $p$  de capacité au moins  $K$ 
6          faire augmenter flot  $f$  le long de  $p$ 
7           $K \leftarrow K/2$ 
8  retourner  $f$ 
```

- c. Montrer que FLOT-MAX-ET-ECHELONNEMENT retourne un flot maximum.
- d. Montrer que la capacité d'une coupe minimum du graphe résiduel G_f vaut au plus $2K|A|$ à chaque exécution de la ligne 4.
- e. Montrer que la boucle **tant que** intérieure des lignes 5–6 est exécutée $O(A)$ fois pour chaque valeur de K .
- f. Conclure que FLOT-MAX-ET-ECHELONNEMENT peut être implémenté de façon à s'exécuter en temps $O(A^2 \lg C)$.

26.6. Flot maximum avec capacités négatives

Supposons qu'un réseau de transport puisse avoir des capacités d'arc négatives (en plus des positives). Dans un tel réseau, il peut très bien ne pas exister de flot réalisable.

- a. On considère un arc (u, v) d'un réseau de transport $G = (S, A)$, où $c(u, v) < 0$. Expliquer brièvement ce que signifie une capacité négative comme celle-là en termes de flot entre u et v .

Soit $G = (S, A)$ un réseau de transport avec capacités d'arc négatives, et soient s et t la source et le puits de G . Construire le réseau de transport ordinaire $G' = (S', A')$ avec la fonction de capacité c' , la source s' et le puits t' , où

$$S' = S \cup \{s', t'\}$$

et

$$\begin{aligned}
 A' = & A \cup \{(u, v) : (v, u) \in A\} \\
 & \cup \{(s', v) : v \in S\} \\
 & \cup \{(u, t') : u \in S\} \\
 & \cup \{(s, t), (t, s)\} .
 \end{aligned}$$

On affecte des capacités aux arcs de la manière suivante. Pour chaque arc $(u, v) \in A$, on pose $c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/2$. Pour chaque sommet $u \in S$, on pose $c'(s', u) = \max(0, (c(S, u) - c(u, S))/2)$ et $c'(u, t') = \max(0, (c(u, S) - c(S, u))/2)$. On pose également $c'(s, t) = c'(t, s) = \infty$.

- b. Démontrer que, s'il existe un flot réalisable dans G , alors toutes les capacités de G' sont positives ou nulles et il existe un flot maximum dans G' tel que tous les arcs menant aux puits t' sont saturés.
- c. Démontrer la proposition inverse de la partie (b). Votre démonstration devra être constructive : étant donné un flot de G' qui sature tous les arcs de t' , il faudra montrer comment obtenir un flot réalisable dans G .
- d. Décrire un algorithme qui trouve un flot réalisable maximum dans G . Soit $MF(|S|, |A|)$ le temps d'exécution le plus défavorable d'un algorithme ordinaire de flot maximum sur un graphe ayant $|S|$ sommets et $|A|$ arcs. Analyser votre algorithme pour calculer le flot maximum d'un réseau de transport à capacités négatives, en fonction de MF .

26.7. Algorithme de couplage biparti de Hopcroft-Karp

Dans ce problème, nous allons décrire un algorithme rapide, dû à Hopcroft et Karp, qui trouve un couplage maximum dans un graphe biparti. L'algorithme s'exécute en temps $O(\sqrt{SA})$. Étant donné un graphe biparti, non orienté $G = (S, A)$, avec $S = L \cup R$ et où toutes les arêtes ont une et une seule extrémité dans L , soit M un couplage de G . On dit qu'une chaîne élémentaire P de G est une **chaîne améliorante** par rapport à M si elle commence en un sommet non couplé de L , s'il finit en un sommet non couplé de R , et si ses arêtes appartiennent alternativement à M et $A - M$. (Cette définition d'une chaîne améliorante est liée à, mais différente de, celle d'une chaîne améliorante dans un réseau de transport.) Dans ce problème, on traite une chaîne comme une séquence d'arêtes, plutôt que comme une séquence de sommets. Une plus courte chaîne améliorante par rapport à un couplage M est une chaîne améliorante ayant un nombre minimum d'arêtes.

Étant donnés deux ensembles A et B , la **différence symétrique** $A \oplus B$ est définie par $(A - B) \cup (B - A)$; c'est donc l'ensemble des éléments qui appartiennent à un seul des deux ensembles.

- a. Montrer que, si M est un couplage et P une chaîne améliorante par rapport à M , alors la différence symétrique $M \oplus P$ est un couplage et $|M \oplus P| = |M| + 1$. Montrer que, si P_1, P_2, \dots, P_k sont des chaînes améliorantes par rapport à M sans sommet commun, alors la différence symétrique $M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ est un couplage de cardinalité $|M| + k$.

La structure générale de notre algorithme est la suivante :

HOPCROFT-KARP(G)

- 1 $M \leftarrow \emptyset$
- 2 **répéter**
- 3 soit $\mathcal{P} \leftarrow \{P_1, P_2, \dots, P_k\}$ un ensemble maximum de plus courts chaînes améliorantes par rapport à M sans sommet commun
- 4 $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$
- 5 **jusqu'à** $\mathcal{P} = \emptyset$
- 6 **retourner** M

Le reste du problème vous demandera d'analyser le nombre d'itérations de l'algorithme (c'est-à-dire, le nombre d'itérations de la boucle **répéter**) et de décrire une implémentation de la ligne 3.

- b.* Étant donnés deux couplages M et M^* de G , montrer que chaque sommet du graphe $G' = (S, M \oplus M^*)$ a un degré au plus égal à 2. En conclure que G' est une union disjointe de chaînes élémentaires ou de cycles. Prouver que les arêtes de chacun de ces chaînes élémentaires ou cycles appartiennent alternativement à M ou M^* . Prouver que, si $|M| \leq |M^*|$, alors $M \oplus M^*$ contient au moins $|M^*| - |M|$ chaînes améliorantes par rapport à M sans sommet commun.

Soit l la longueur d'une plus courte chaîne améliorante par rapport à un couplage M , et soient P_1, P_2, \dots, P_k un ensemble maximum de chaînes améliorantes de longueur l par rapport à M , sans sommet commun. Soit $M' = M \oplus (P_1 \cup \dots \cup P_k)$, et supposons que P soit une plus courte chaîne améliorante par rapport à M' .

- c.* Montrer que, si P n'a pas de sommet commun avec P_1, P_2, \dots, P_k , alors P a plus de l arête.
- d.* Supposons maintenant que P ait au moins un sommet commun avec P_1, P_2, \dots, P_k . Soit B l'ensemble des arêtes $(M \oplus M') \oplus P$. Montrer que $B = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$ et que $|B| \geq (k+1)l$. En conclure que P a plus de l arête.
- e.* Prouver que, si une plus courte chaîne améliorante par rapport à M a l arêtes, alors la taille du couplage maximum est au plus égale à $|M| + |S| / (l+1)$.
- f.* Montrer que le nombre d'itérations de la boucle **répéter** de l'algorithme est au plus égal à $2\sqrt{S}$. (*conseil* : De combien peut croître M après l'itération numéro \sqrt{S} ?)
- g.* Donner un algorithme à temps $O(A)$ pour trouver un ensemble maximum de plus courtes chaînes améliorantes sans sommet commun P_1, P_2, \dots, P_k pour un couplage M donné. En conclure que le temps d'exécution total de HOPCROFT-KARP est $O(\sqrt{S}A)$.

NOTES

Ahuja, Magnanti et Orlin [7], Even [87], Lawler [196], Papadimitriou et Steiglitz [237], et Tarjan [292] sont de bonnes références pour les flots et les algorithmes afférents. Goldberg, Tardos et Tarjan [119] offre une présentation intéressante des algorithmes pour problèmes de flot, et Schrijver [267] a écrit une étude passionnante qui retrace l'histoire des découvertes dans le domaine des flots en réseau.

La méthode de Ford-Fulkerson est due à Ford et Fulkerson [93], qui furent à l'origine de nombreux problèmes touchant aux flots, notamment les problèmes du flot maximum et du couplage biparti. Les premières implémentations de la méthode de Ford-Fulkerson trouvaient souvent les chemins améliorants par une recherche en largeur ; Edmonds et Karp [86], et de manière indépendante Dinic [76], ont montré que cette stratégie générait un algorithme à temps polynomial. Une idée voisine, basée sur l'emploi de « flots bloquants », est également due à Dinic [76]. Karzanov [176] eut le premier l'idée des préflots. La méthode de préflot est due à Goldberg [117] et Goldberg et Tarjan [121]. Goldberg et Tarjan ont donné un algorithme à temps $O(S^3)$ qui emploie une file pour gérer l'ensemble des sommets débordants, ainsi qu'un algorithme qui utilise des arbres dynamiques pour atteindre un temps de $O(SA \lg(S^2/A + 2))$. Plusieurs autres chercheurs ont inventé des algorithmes de flot maximum du genre préflot. Ahuja et Orlin [9] et Ahuja, Orlin et Tarjan [10] ont donné des algorithmes basés sur l'échelonnement. Cheriyan et Maheshwari [55] ont proposé de pousser le flot depuis le sommet débordant de hauteur maximale. Cheriyan et Hagerup [54] ont suggéré de permuter aléatoirement les liste de voisinage, et plusieurs chercheurs [14, 178, 241] ont inventé des dérandomisations astucieuses de cette idée, ce qui a engendré une série d'algorithmes rapides. L'algorithme de King, Rao et Tarjan [178], qui est l'algorithme le plus rapide de cette catégorie, s'exécute en $O(SA \log_{A/(S \lg S)} S)$.

À ce jour, l'algorithme le plus rapide asymptotiquement pour le problème du flot maximum est dû à Goldberg et Rao [120] et il s'exécute en temps $O(\min(S^{2/3}, A^{1/2}) A \lg(S^2/A + 2) \lg C)$, où $C = \max_{(u,v) \in A} c(u, v)$. Cet algorithme n'emploie pas la méthode de préflot, mais recherche des flots bloquants. Tous les algorithmes de flot maximum antérieurs, y compris ceux de ce chapitre, font appel à une certaine notion de distance (ou hauteur) et assignent implicitement à chaque arc une longueur de 1. Cet nouvel algorithme adopte une approche différente : il assigne une longueur de 0 aux arcs de haute capacité et une longueur de 1 aux arcs de faible capacité. Disons, de manière informelle, que, avec ces longueurs, les plus courts chemins de la source au puits ont tendance à avoir une forte capacité, ce qui signifie que l'on a besoin de moins d'itérations.

En pratique, les algorithmes de préflot l'emportent sur les algorithmes à chemins améliorants ou sur la programmation linéaire, pour ce qui concerne le problème du flot maximum. Une étude faite par Cherkassky et Goldberg [56] souligne l'importance de deux heuristiques pour la mise en œuvre d'un algorithme de préflot. La première heuristique consiste à effectuer périodiquement une recherche en largeur du graphe résiduel, afin d'obtenir des valeurs de hauteur plus précises. La seconde heuristique est celle du fossé (voir exercice 26.5.5 de préflot). Les auteurs concluent que la meilleure variante de préflot est celle qui choisit de décharger le sommet débordant ayant la hauteur maximale.

À ce jour, le meilleur algorithme pour couplage biparti maximum, découvert par Hopcroft et Karp [152], s'exécute en temps $O(\sqrt{SA})$ (voir problème 26.5.12). Le livre de Lovász et Plummer [207] est une excellente référence en matière de problèmes de couplage.

PARTIE 7

MORCEAUX CHOISIS

Cette partie contient une sélection de sujets d'algorithmique qui étendent et complètent les chapitres précédents. Certains chapitres introduisent de nouveaux modèles de calcul, comme les réseaux de tri ou le calcul matriciel. D'autres couvrent des domaines spécialisés comme la géométrie algorithmique ou la théorie des nombres. Les deux derniers chapitres étudient certaines limitations connues en matière de conception d'algorithmes efficaces, et introduisent des techniques permettant de faire face à ces limitations.

Le chapitre 27 présente un modèle de calcul parallèle : les réseaux de comparaison. *Grosso modo*, un réseau de comparaison est un algorithme qui permet d'effectuer plusieurs comparaisons simultanément. Ce chapitre montre comment construire un réseau de comparaison capable de trier n nombres en $O(\lg^2 n)$.

Le chapitre 28 étudie des algorithmes efficaces de traitement des matrices. Après avoir présenté quelques propriétés fondamentales des matrices, il étudie l'algorithme de Strassen, qui peut multiplier deux matrices $n \times n$ en $O(n^{2,81})$. Il présente ensuite deux méthodes générales, les décompositions LU et les décompositions LUP, pour résoudre les équations linéaires, via élimination de Gauss, en $O(n^3)$. Il montre aussi qu'il est possible d'effectuer inversion et multiplication matricielle avec des performances identiques. Le chapitre se termine en montrant comment calculer une solution approchée par la méthode des moindres carrés quand un ensemble d'équations linéaires n'a pas de solution exacte.

Le chapitre 29 présente la programmation linéaire qui permet de maximiser ou minimiser un objectif, compte tenu de ressources limitées et de contraintes contradictoires. La programmation linéaire revient dans toutes sortes de domaines concrets. Ce chapitre montre comment formuler et résoudre les programmes linéaires. La méthode de résolution traitée ici est l'algorithme du simplexe, qui est le plus vieil algorithme de programmation linéaire. Comparé à beaucoup d'autres algorithmes de ce livre, l'algorithme du simplexe ne s'exécute pas en temps polynomial dans le cas le plus défavorable ; mais, en pratique, il est assez efficace et très utilisé.

Le chapitre 31 étudie les opérations sur les polynômes, et montre qu'une technique bien connue en traitement du signal, la transformée rapide de Fourier, peut servir à multiplier deux polynômes de degré n en $O(n \lg n)$. Il étudie aussi des implémentations efficaces de la transformée rapide de Fourier, parmi lesquelles un circuit parallèle.

Le chapitre 31 présente des algorithmes de la théorie des nombres. Après une révision succincte de la théorie des nombres, il expose l'algorithme d'Euclide servant à calculer les plus grands communs diviseurs. Sont ensuite présentés des algorithmes de résolution d'équations linéaires de congruence et d'élévation d'un nombre à une puissance modulo un autre nombre. Suit une application majeure des algorithmes de la théorie des nombres : le crypto-système à clé publique RSA. Ce crypto-système permet non seulement de chiffrer des messages, mais aussi de créer des signatures numériques. Le chapitre présente ensuite le test de primalité randomisé de Miller-Rabin qui peut servir à trouver efficacement de grands nombres premiers, ce qui est une exigence majeure du système RSA. Enfin, le chapitre traite de l'heuristique « rho » de Pollard pour factoriser les entiers et discute de l'état de l'art en matière de factorisation de nombres entiers.

Le chapitre 32 étudie le problème consistant à trouver toutes les occurrences d'une chaîne donnée dans un texte donné, problème qui apparaît fréquemment dans les programmes d'édition de texte. Après étude de la méthode naïve, le chapitre donne une méthode élégante due à Rabin et Karp. Ensuite, après avoir donné une solution efficace basée sur les automates finis, le chapitre présente l'algorithme de Knuth-Morris-Pratt dont l'efficacité repose sur un pré traitement astucieux de la chaîne à rechercher.

La géométrie algorithmique est le sujet du chapitre 33. Après une étude des primitives élémentaires de la géométrie algorithmique, le chapitre montre comment une méthode de « balayage » peut déterminer efficacement si un ensemble de segments de droite contient des intersections. Deux algorithmes astucieux permettant de trouver l'enveloppe convexe d'un ensemble de points, le balayage de Graham et la marche de Jarvis, illustrent également la puissance des méthodes de balayage. Le chapitre se termine par un algorithme efficace permettant de trouver le couple de points qui sont les plus proches mutuellement parmi un ensemble de points du plan.

Le chapitre 34 concerne les problèmes NP-complets. Nombre de problèmes calculatoires intéressants sont NP-complets, mais l'on ne connaît aucun algorithme en

temps polynomial pour les résoudre. Ce chapitre présente des techniques permettant d'établir la NP-complétude d'un problème. Il prouvera que plusieurs problèmes classiques sont NP-complets : déterminer si un graphe a un circuit hamiltonien ; déterminer si une formule booléenne est satisfaisable ; déterminer si un ensemble donné de nombres possède un sous-ensemble dont la somme est égale à une valeur cible donnée. Le chapitre montrera également que le fameux problème du voyageur de commerce est NP-complet.

Le chapitre 35 montre comment utiliser des algorithmes d'approximation pour trouver efficacement des solutions approchées pour les problèmes NP-complets. Pour certains problèmes, il est assez facile de produire des solutions approchées qui sont quasiment optimales. Pour d'autres problèmes, même les meilleurs algorithmes d'approximation connus faiblissent à mesure que la taille du problème augmente. Ensuite, il existe des problèmes pour lesquels l'amélioration continue de l'approximation s'accompagne d'un allongement continu du temps de calcul. Ce chapitre illustre ces possibilités sur les problèmes suivants : couverture de sommets (versions non pondérée et pondérée) ; version d'optimisation de la satisfaisabilité 3-CNF ; problème du voyageur de commerce ; couverture d'ensemble ; somme de sous-ensembles.

Chapitre 27

Réseaux de tri

Dans la partie 2, nous avons examiné des algorithmes de tri pour ordinateurs séquentiels (machines à accès aléatoire) qui n'autorisent qu'une opération à la fois. Dans ce chapitre, nous allons étudier des algorithmes de tri basés sur un modèle de calcul dit « réseau de comparaison », dans lequel plusieurs opérations de comparaison peuvent être effectuées simultanément.

Les réseaux de comparaison diffèrent des ordinateurs séquentiels sur deux points essentiels. D'abord, il ne savent effectuer que des comparaisons. Il est donc impossible d'implémenter un algorithme comme le tri par dénombrement (voir section 8.2) sur un réseau de comparaison. Ensuite, contrairement au modèle séquentiel où les opérations se produisent en série (l'une après l'autre), les opérations d'un réseau de comparaison peuvent s'exécuter en même temps, « en parallèle » comme on dit. Comme on le verra, cette caractéristique permet de construire des réseaux de comparaison qui trient n valeurs dans un temps infra-linéaire.

On commence à la section 27.1 par définir les réseaux de comparaison et les réseaux de tri. On donne également une définition naturelle du « temps d'exécution » d'un réseau de comparaison en fonction de la profondeur du réseau. La section 27.2 démontre le « principe du zéro-un » qui facilite grandement l'analyse de la validité des réseaux de tri.

Le réseau de tri efficace que nous allons mettre au point est essentiellement une version parallèle de l'algorithme de tri par fusion de la section 2.3.1. Notre construction comprendra trois étapes. La section 27.3 présentera la conception d'une trieuse « bitonique » qui constituera notre première pierre. Nous modifierons légèrement la

trieuse bitonique à la section 27.4 pour produire un réseau de fusion capable de fusionner deux séquences triées en une séquence triée unique. Enfin, à la section 27.5, ces réseaux de fusion seront assemblés en un réseau de tri capable de trier n valeurs en $O(\lg^2 n)$.

27.1 RÉSEAUX DE COMPARAISON

Les réseaux de tri étant des réseaux de comparaison qui trient systématiquement leurs entrées, il est utile de commencer notre étude avec les réseaux de comparaison et leurs caractéristiques. Un réseau de comparaison est uniquement composé de câbles et de comparateurs. Un *comparateur*, comme celui de la figure 27.1(a), est un dispositif à deux entrées, x et y , et deux sorties, x' et y' , qui réalise la fonction suivante :

$$\begin{aligned}x' &= \min(x, y) , \\y' &= \max(x, y) .\end{aligned}$$

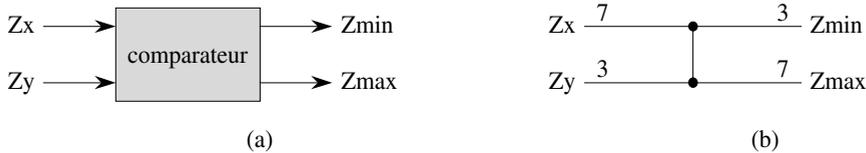


Figure 27.1 (a) Un comparateur avec les entrées x et y et les sorties x' et y' . (b) Le même comparateur, représenté par un seul trait vertical. On peut voir les entrées $x = 7, y = 3$ et les sorties $x' = 3, y' = 7$.

La représentation graphique de la figure 27.1(a) étant trop encombrante, nous adopterons la convention consistant à dessiner les comparateurs comme des segments de droite verticaux, comme sur la figure 27.1(b). Les entrées apparaissent à gauche et les sorties à droite, la plus petite valeur d'entrée apparaissant sur la sortie du haut, et la plus grande valeur d'entrée apparaissant sur la sortie du bas. On peut donc dire d'un comparateur qu'il trie ses deux entrées.

On supposera que l'action de chaque comparateur prend $O(1)$. Autrement dit, on suppose que le temps entre l'apparition des valeurs d'entrée x et y et la production des valeurs de sortie x' et y' est une constante.

Un *câble* transmet une valeur d'un endroit à l'autre. Les câbles peuvent relier la sortie d'un comparateur à l'entrée d'un autre, mais autrement ce sont soit des câbles d'entrée de réseau, soit des câbles de sortie de réseau. Tout au long de ce chapitre, on supposera qu'un réseau de comparaison contient n *câbles d'entrée* a_1, a_2, \dots, a_n , à travers lesquels les valeurs à trier arrivent dans le réseau, et n *câbles de sortie* b_1, b_2, \dots, b_n , qui produisent les résultats calculés par le réseau. On parlera également de *séquence d'entrée* $\langle a_1, a_2, \dots, a_n \rangle$ et de *séquence de sortie* $\langle b_1, b_2, \dots, b_n \rangle$,

pour faire référence aux valeurs circulant sur les câbles d'entrée et de sortie. Autrement dit, on utilisera le même nom pour le câble et la valeur qu'il transporte. Le contexte permettra de ne pas faire d'erreur d'interprétation.

La figure 27.2 montre un *réseau de comparaison*, qui est un ensemble de comparateurs reliés par des câbles. On dessine un réseau de comparaison à n entrées comme une collection de n *lignes* horizontales sur lesquelles sont placés verticalement des comparateurs. On remarquera qu'une ligne ne représente *pas* un câble unique, mais bien une séquence de câbles distincts reliant divers comparateurs. Par exemple, la ligne du haut sur la figure 27.2 représente trois câbles : le câble d'entrée a_1 , qui relie à une entrée du comparateur A ; un câble reliant la sortie supérieure du comparateur A à une entrée du comparateur C ; et le câble de sortie b_1 , qui vient de la sortie supérieure du comparateur C . Chaque entrée de comparateur est reliée à un câble qui est soit l'un des n câbles d'entrée a_1, a_2, \dots, a_n du réseau, soit connecté à la sortie d'un autre comparateur. De même, chaque sortie de comparateur est reliée à un câble qui est soit l'un des n câbles de sortie b_1, b_2, \dots, b_n du réseau, soit est relié à l'entrée d'un autre comparateur. La contrainte principale pour les comparateurs interconnectés est que le graphe des interconnexions soit acyclique : si l'on suit un chemin reliant la sortie d'un comparateur à l'entrée d'un autre, à une sortie, à une entrée, etc. on ne doit jamais revenir sur ses pas et passer deux fois par le même comparateur. Donc, comme dans la figure 27.2, on peut dessiner un réseau de comparaison avec les entrées de réseau à gauche et les sorties à droites ; les données se déplacent à travers le réseau de la gauche vers la droite.

Chaque comparateur ne produit ses valeurs de sortie que lorsque ses deux valeurs d'entrée sont disponibles. Dans la figure 27.2(a), par exemple, supposons que la séquence $\langle 9, 5, 2, 6 \rangle$ apparaisse sur les câbles d'entrée au temps 0. Au temps 0, donc, seuls les comparateurs A et B ont reçu leurs valeurs d'entrée. En admettant que chaque comparateur nécessite une unité de temps pour calculer ses valeurs de sortie, les comparateurs A et B produisent leurs sorties au temps 1 ; les valeurs résultantes sont montrées à la figure 27.2(b). On notera que les comparateurs A et B produisent leurs valeurs en même temps, ou comme on dit « en parallèle ». Maintenant, au temps 1, les comparateurs C et D , disposent de leurs valeurs d'entrée, mais pas E . Une unité de temps plus tard, au temps 2, ils produisent leurs sorties, comme le montre la figure 27.2(c). Les comparateurs C et D agissent également en parallèle. La sortie supérieure du comparateur C et la sortie inférieure du comparateur D sont respectivement reliées aux câbles de sortie b_1 et b_4 du réseau de comparaison, et ces câbles de sortie transportent donc leurs valeurs finales au temps 2. Entre-temps, au temps 2, le comparateur E a reçu ses entrées, et la figure 27.2(d) montre qu'il produit ses valeurs de sortie au temps 3. Ces valeurs sont transportées par les câbles de sortie de réseau b_2 et b_3 , et la séquence de sortie $\langle 2, 5, 6, 9 \rangle$ est maintenant complète.

En gardant l'hypothèse d'un temps unitaire pour l'action de chaque comparateur, on peut définir le « temps d'exécution » d'un réseau de comparaison, c'est-à-dire le temps nécessaire pour que les câbles de sortie reçoivent leur valeurs après que les

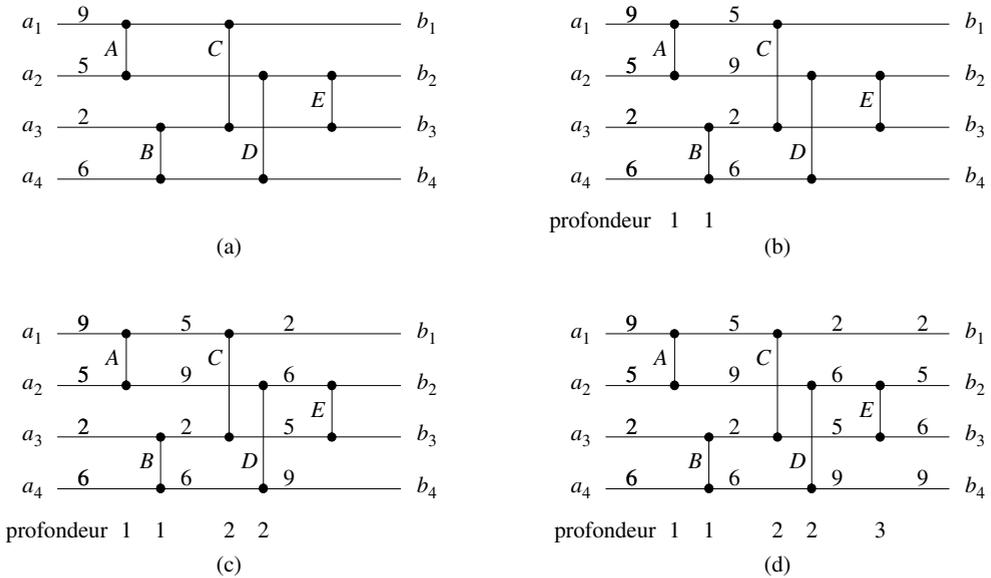


Figure 27.2 (a) Un réseau de comparaison à 4 entrées et 4 sorties, qui est en réalité un réseau de tri. Au temps 0, les valeurs d'entrées apparaissent sur les quatre câbles d'entrée. (b) Au temps 1, les valeurs représentées apparaissent aux sorties des comparateurs A et B, qui se trouvent à la profondeur 1. (c) Au temps 2, les valeurs apparaissent aux sorties des comparateurs C et D, situées à la profondeur 2. Les câbles de sortie b_1 et b_4 ont maintenant leurs valeurs finales, ce qui n'est pas encore le cas de b_2 et b_3 . (d) Au temps 3, les valeurs apparaissent aux sorties du comparateur E, à la profondeur 3. Les câbles de sortie b_2 et b_3 ont maintenant leurs valeurs finales.

câbles d'entrée ont reçu les leurs. De manière informelle, ce temps est égal au plus grand nombre de comparateurs à travers lequel peut passer un élément d'entrée pendant qu'il voyage entre un câble d'entrée et un câble de sortie. Plus formellement, on définit la **profondeur** d'un câble de la manière suivante. Un câble d'entrée d'un réseau de comparaison possède la profondeur 0. Ensuite, si un comparateur possède deux câbles d'entrée de profondeurs d_x et d_y , alors ses câbles de sortie ont la profondeur $\max(d_x, d_y) + 1$. Comme il n'existe aucun cycle de comparateurs dans un réseau de comparaison, la profondeur d'un câble est bien définie, et on définit la profondeur d'un comparateur comme étant la profondeur de ses câbles de sortie. La figure 27.2 montre les profondeurs des comparateurs. La profondeur d'un réseau de comparaison est la profondeur maximale d'un câble de sortie ou, si l'on préfère, la profondeur maximale d'un comparateur. Le réseau de comparaison de la figure 27.2, par exemple, a une profondeur 3 parce que le comparateur E a la profondeur 3. Si chaque comparateur dépense une unité de temps pour produire sa valeur de sortie, et si les entrées du réseau apparaissent au temps 0, alors un comparateur à la profondeur d produit ses sorties au temps d ; la profondeur du réseau est donc égale au temps nécessaire pour que le réseau produise les valeurs de tous ses câbles de sortie.

Un *réseau de tri* est un réseau de comparaison pour lequel la séquence de sortie est monotone croissante ($b_1 \leq b_2 \leq \dots \leq b_n$) pour toute séquence d'entrée. Bien entendu, tous les réseaux de comparaison ne sont pas des réseaux de tri, mais celui de la figure 27.2 en est un. Pour comprendre pourquoi, on remarque qu'après le temps 1, le minimum des quatre valeurs d'entrée a été placé soit dans la sortie supérieure du comparateur *A*, soit dans la sortie supérieure du comparateur *B*. Après le temps 2, il doit donc se trouver sur la sortie supérieure du comparateur *C*. Un raisonnement symétrique montre qu'après le temps 2, le maximum des quatre valeurs d'entrée s'est retrouvé sur la sortie inférieure du comparateur *D*. Tout ce qui reste alors à faire, c'est que le comparateur *E* s'arrange pour que les deux valeurs intermédiaires se retrouvent aux bonnes positions de sortie, ce qui se produit au temps 3.

Un réseau de comparaison ressemble à une procédure en ce sens qu'il spécifie la façon dont les comparaisons doivent se faire, mais il diffère d'une procédure en ce sens que sa *taille* (nombre de comparateurs qu'il contient) dépend du nombre d'entrées et de sorties. Ce que nous allons décrire, c'est en réalité des « familles » de réseaux de comparaison. Par exemple, le but de ce chapitre est de développer une famille TRIEUSE de réseaux de tri efficaces. On spécifie un réseau particulier d'une famille par le nom de la famille et le nombre d'entrées (qui est égal au nombre de sorties). Ainsi, le réseau de tri à n entrées et n sorties de la famille TRIEUSE s'appellera TRIEUSE[n].

Exercices

27.1.1 Montrer les valeurs qui apparaissent sur tous les câbles du réseau de la figure 27.2 lorsqu'on lui donne la séquence d'entrée $\langle 9, 6, 5, 2 \rangle$.

27.1.2 Soit n une puissance exacte de 2. Montrer comment construire un réseau de comparaison à n entrées et n sorties de profondeur $\lg n$, pour lequel le câble de sortie supérieur transporte toujours la plus petite valeur d'entrée et le câble de sortie inférieur transporte toujours la plus grande valeur d'entrée.

27.1.3 On peut prendre un réseau de tri et y ajouter un comparateur, pour obtenir un réseau de comparaison qui n'est pas un réseau de tri. Montrer comment ajouter un comparateur au réseau de la figure 27.2 de telle façon que le réseau résultant ne trie pas toutes les permutations de l'entrée.

27.1.4 Démontrer qu'un réseau de tri quelconque sur n entrées a une profondeur au moins égale à $\lg n$.

27.1.5 Démontrer que le nombre de comparateurs d'un réseau de tri quelconque est $\Omega(n \lg n)$.

27.1.6 On considère le réseau de comparaison de la figure 27.3. Démontrer que c'est en réalité un réseau de tri, et expliquer en quoi sa structure est apparentée à celle du tri par insertion (section 2.1).

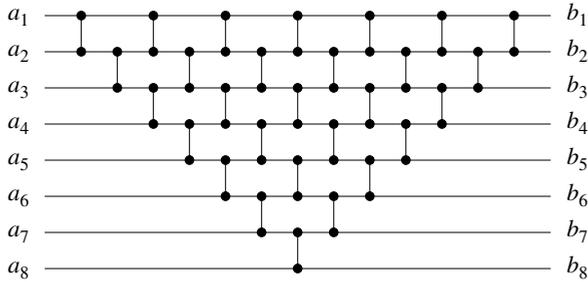


Figure 27.3 Un réseau de tri basé sur le tri par insertion, servant de support à l'exercice 27.1.6.

27.1.7 On peut représenter un réseau de comparaison à n entrées et c comparateurs comme une liste de c paires d'entiers de l'intervalle 1 à n . Si deux paires contiennent un entier en commun, l'ordre des comparateurs correspondant dans le réseau est déterminé par celui des paires dans la liste. À l'aide de cette représentation, décrire un algorithme (séquentiel) en $O(n + c)$ pour déterminer la profondeur d'un réseau de comparaison.

27.1.8 * Supposons que, en plus du comparateur standard, on ait un comparateur « inversé » qui produise sa sortie minimale sur le câble inférieur et sa sortie maximale sur le câble supérieur. Montrer comment convertir un réseau de tri qui utilise un total de c comparateurs (standard et inversés) en un réseau qui utilise c comparateurs standard. Prouver la validité de la méthode de conversion.

27.2 LE PRINCIPE DU ZÉRO-UN

Le *principe du zéro-un* dit que, si un réseau de tri fonctionne correctement lorsque chaque entrée est prise dans l'ensemble $\{0, 1\}$, alors il fonctionne correctement sur des nombres arbitraires. (Les nombres peuvent être entiers, réels ou, plus généralement, n'importe quel ensemble de valeurs prises dans un ensemble linéairement ordonné quelconque.) Quand on construit des réseaux de tri et d'autres réseaux de comparaison, le principe du zéro-un permet de ne s'intéresser qu'à leur action pour des séquences d'entrées composées uniquement de 0 et de 1. Une fois qu'un réseau de tri est construit et qu'on a démontré qu'il pouvait trier toutes les séquences de zéros et de uns, on s'appuie sur ce principe pour montrer qu'il trie correctement des séquences de valeurs arbitraires.

La démonstration du principe du zéro-un s'appuie sur la notion de fonction monotone croissante (section 3.2).

Lemme 27.1 Si un réseau de comparaison transforme la séquence d'entrée $a = \langle a_1, a_2, \dots, a_n \rangle$ en la séquence de sortie $b = \langle b_1, b_2, \dots, b_n \rangle$, alors pour une fonction f monotone croissante quelconque, le réseau transforme la séquence d'entrée $f(a) = \langle f(a_1), f(a_2), \dots, f(a_n) \rangle$ en la séquence de sortie $f(b) = \langle f(b_1), f(b_2), \dots, f(b_n) \rangle$.

Démonstration : Nous allons démontrer en premier lieu que, si f est une fonction monotone croissante, alors un comparateur unique recevant les entrées $f(x)$ et $f(y)$ produit les sorties $f(\min(x, y))$ et $f(\max(x, y))$. Le lemme sera ensuite démontré par récurrence.

Considérons un comparateur dont les valeurs d'entrée sont x et y . La sortie supérieure du comparateur est $\min(x, y)$ et la sortie inférieure est $\max(x, y)$. Supposons que la fonction f soit appliquée aux entrées du comparateurs, donnant ainsi les nouvelles entrées $f(x)$ et $f(y)$, comme à la figure 27.4. L'action du comparateur engendre la valeur $\min(f(x), f(y))$ sur la sortie supérieure, et $\max(f(x), f(y))$ sur la sortie inférieure. Comme f est monotone croissante, $x \leq y$ implique $f(x) \leq f(y)$. On obtient donc les identités suivantes :

$$\begin{aligned}\min(f(x), f(y)) &= f(\min(x, y)), \\ \max(f(x), f(y)) &= f(\max(x, y)).\end{aligned}$$

Le comparateur produit donc les valeurs $f(\min(x, y))$ et $f(\max(x, y))$ à partir des entrées $f(x)$ et $f(y)$, ce qui termine la démonstration.

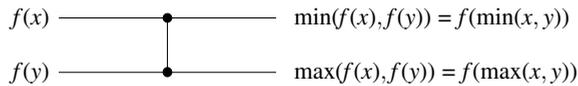


Figure 27.4 L'action du comparateur dans la démonstration du lemme 27.1. La fonction f est monotone croissante.

On peut utiliser une récurrence sur la profondeur de chaque câble dans un réseau de comparaison général, pour prouver un résultat plus fort que celui énoncé dans le lemme : si un câble prend la valeur a_i lorsque la séquence d'entrée a est appliquée au réseau, alors il prend la valeur $f(a_i)$ quand on applique la séquence d'entrée $f(a)$. Comme les câbles de sortie sont inclus dans l'énoncé, démontrer ce résultat démontrera le lemme par la même occasion.

Pour la base, considérons un câble de profondeur 0, c'est-à-dire un câble d'entrée a_i . Le résultat s'en déduit de façon triviale : quand $f(a)$ est appliquée au réseau, le câble d'entrée transporte $f(a_i)$. Pour l'étape inductive, on considère un câble de profondeur $d \geq 1$. Le câble est la sortie d'un comparateur de profondeur d , et les câbles d'entrée de ce comparateur ont une profondeur strictement inférieure à d . Par conséquent, d'après l'hypothèse de récurrence, si les câbles d'entrée du comparateur transportent les valeurs a_i et a_j quand la séquence d'entrée a est appliquée, alors ils transportent $f(a_i)$ et $f(a_j)$ quand la séquence d'entrée $f(a)$ est appliquée. D'après notre affirmation précédente, les câbles de sortie de ce comparateur transportent alors $f(\min(a_i, a_j))$ et $f(\max(a_i, a_j))$. Puisqu'ils transportent $\min(a_i, a_j)$ et $\max(a_i, a_j)$ quand la séquence d'entrée est a , le lemme est démontré. \square

En guise d'exemple d'application du lemme 27.1, la figure 27.5(b) montre le réseau de tri de la figure 27.2 (repris sur la figure 27.5(a)) quand on applique la fonction monotone croissante $f(x) = \lceil x/2 \rceil$ à ses entrées. La valeur transportée par chaque câble est l'image par f de la valeur transportée par le même câble sur la figure 27.2.

Quand un réseau de comparaison est un réseau de tri, le lemme 27.1 permet de démontrer le résultat remarquable suivant :

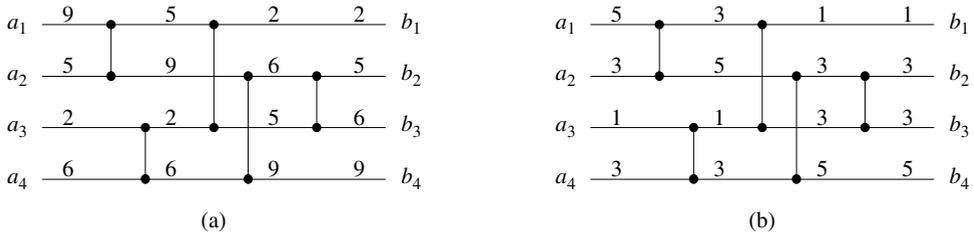


Figure 27.5 (a) Le réseau de tri de la figure 27.2 avec la séquence d'entrée $\langle 9, 5, 2, 6 \rangle$. (b) Le même réseau de tri quand on applique à ses entrées la fonction $f(x) = \lceil x/2 \rceil$. Chaque câble de ce réseau porte l'image par f de la valeur portée par le câble correspondant en (a).

Théorème 27.2 (Principe du zéro-un) *Si un réseau de comparaison à n entrées trie les 2^n séquences possibles de 0 et de 1 correctement, alors il trie correctement toutes les séquences de nombres arbitraires.*

Démonstration : Supposons, en raisonnant par l'absurde, que le réseau trie toutes les séquences zéro-un, mais qu'il existe une séquence de nombres arbitraires que le réseau ne trie pas correctement. Autrement dit, il existe une séquence d'entrée $\langle a_1, a_2, \dots, a_n \rangle$ contenant des éléments a_i et a_j tels que $a_i < a_j$ mais qui seront dans l'ordre inverse dans la séquence de sortie. On définit une fonction f monotone croissante par

$$f(x) = \begin{cases} 0 & \text{si } x \leq a_i, \\ 1 & \text{si } x > a_i. \end{cases}$$

Puisque le réseau place a_j avant a_i dans la séquence de sortie quand $\langle a_1, a_2, \dots, a_n \rangle$ est l'entrée, le lemme 27.1 dit qu'il place $f(a_j)$ avant $f(a_i)$ dans la séquence de sortie quand l'entrée est $\langle f(a_1), f(a_2), \dots, f(a_n) \rangle$. Mais comme $f(a_j) = 1$ et $f(a_i) = 0$, cela signifie que le réseau ne peut pas trier la séquence zéro-un $\langle f(a_1), f(a_2), \dots, f(a_n) \rangle$ correctement, ce qui contredit l'hypothèse de départ. \square

Exercices

27.2.1 Démontrer que l'application d'une fonction monotone croissante à une séquence triée produit une séquence triée.

27.2.2 Démontrer qu'un réseau de comparaison à n entrées trie correctement la séquence d'entrée $\langle n, n - 1, \dots, 1 \rangle$ si et seulement si il est capable de trier correctement les $n - 1$ séquences zéro-un $\langle 1, 0, 0, \dots, 0, 0 \rangle, \langle 1, 1, 0, \dots, 0, 0 \rangle, \dots, \langle 1, 1, 1, \dots, 1, 0 \rangle$.

27.2.3 Utiliser le principe du zéro-un pour prouver que le réseau de comparaison de la figure 27.6 est un réseau de tri.

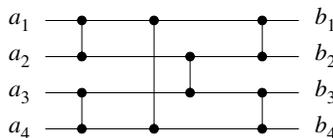


Figure 27.6 Un réseau de comparaison pour trier 4 nombres.

27.2.4 Énoncer et démontrer un principe analogue à celui du zéro-un pour un modèle d'arbre de décision. (*conseil* : Attention à gérer correctement l'égalité.)

27.2.5 Démontrer qu'un réseau de tri à n entrées doit contenir au moins un comparateur entre les i ème et $(i + 1)$ ème lignes pour tout $i = 1, 2, \dots, n - 1$.

27.3 UN RÉSEAU DE TRI BITONIQUE

La première étape de notre construction d'un réseau de tri efficace consistera à mettre au point un réseau de comparaison capable de trier une **séquence bitonique** quelconque : une séquence qui croît de façon monotone puis décroît de façon monotone, ou qui peut subir une rotation circulaire pour devenir monotone croissante puis monotone décroissante. Par exemple, les séquences $\langle 1, 4, 6, 8, 3, 2 \rangle$, $\langle 6, 9, 4, 2, 3, 5 \rangle$ et $\langle 9, 8, 3, 2, 4, 6 \rangle$ sont toutes bitoniques. Comme condition aux limites, nous dirons qu'une séquence composée de 1 ou 2 nombres est bitonique. Les séquences zéro-un qui sont bitoniques ont une structure simple. Elles ont soit la forme $0^i 1^j 0^k$, soit la forme $1^i 0^j 1^k$, pour $i, j, k \geq 0$. Notez qu'une séquence qui est soit monotone croissante, soit monotone décroissante, est également bitonique.

La trieuse bitonique que nous allons construire est un réseau de comparaison capable de trier des séquences bitoniques composées de 0 et de 1. L'exercice 27.3.6 demandera de montrer que la trieuse bitonique peut trier des séquences bitoniques de nombres arbitraires.

a) Le séparateur

Une trieuse bitonique est composée de plusieurs étages, qui portent le nom de **séparateurs**. Chaque séparateur est un réseau de comparaison de profondeur 1 dans lequel la ligne d'entrée i est comparée avec la ligne $i + n/2$ pour $i = 1, 2, \dots, n/2$. (On suppose que n est pair.) La figure 27.7 montre SÉPARATEUR[8], le séparateur à 8 entrées et 8 sorties.

Quand une séquence bitonique de 0 et de 1 est placée en entrée d'un séparateur, celui-ci produit une séquence de sortie dans laquelle les valeurs plus faibles se trouvent dans la moitié supérieure, et les valeurs plus grandes se trouvent dans la moitié inférieure, les deux moitiés étant elles-mêmes des séquences bitoniques. En fait, au moins une moitié est **pure**, c'est-à-dire composée uniquement de 0 ou de 1. (Notez que toutes les séquences pures sont bitoniques.) Le lemme suivant établit ces propriétés des séparateurs.

Lemme 27.3 *Si l'entrée d'un séparateur est une séquence bitonique de 0 et de 1, alors la sortie vérifie les propriétés suivantes : les moitiés haute et basse sont toutes les deux bitoniques, tout élément de la moitié supérieure est au moins aussi petit que tout élément de la moitié inférieure, et l'une au moins des moitiés est pure.*

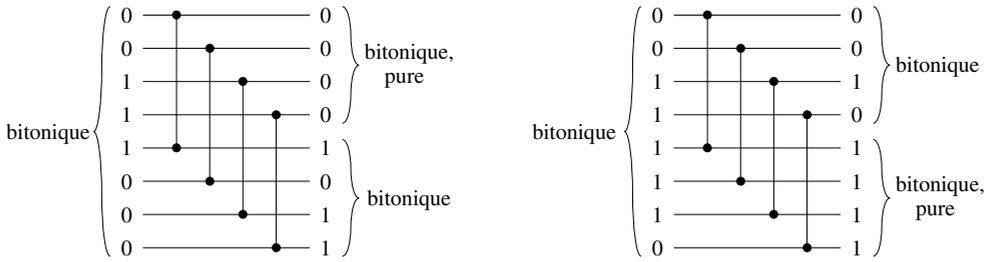


Figure 27.7 Le réseau de comparaison SÉPARATEUR[8]. Deux exemples de valeurs d'entrée et de sortie zéro-un sont proposés. On suppose que l'entrée est bitonique. Un séparateur garantit que tout élément de sortie de la moitié supérieure est inférieur ou égal à tout élément de sortie de la moitié inférieure. Par ailleurs, les deux moitiés sont bitoniques et une moitié au moins est pure.

Démonstration : Le réseau de comparaison SÉPARATEUR[n] compare les entrées i et $i + n/2$ pour $i = 1, 2, \dots, n/2$. On peut, sans remettre en cause la généralité de la démonstration, supposer que l'entrée est de la forme $00 \dots 011 \dots 100 \dots 0$. (La situation où l'entrée est de la forme $11 \dots 100 \dots 011 \dots 1$ est symétrique.) Il existe trois cas de figure possibles, selon que le point milieu $n/2$ se trouve dans un bloc de 0 ou de 1, et l'un de ces cas (celui pour lequel le point milieu apparaît dans un bloc de 1) se divise lui-même en deux cas. Les quatre cas sont représentés sur la figure 27.8. Pour chacun des cas montrés, le lemme est vérifié. □

b) La trieuse bitonique

En combinant récursivement des séparateurs, comme dans la figure 27.9, on peut construire une **trieuse bitonique**, qui est un réseau capable de trier des séquences bitoniques. Le premier palier de TRIEUSE-BITONIQUE[n] est constitué d'un SÉPARATEUR[n] qui, d'après le lemme 27.3, produit deux séquences bitoniques de taille moitié moindre telles que tout élément de la moitié supérieure soit au moins aussi petit que tout élément de la moitié basse. On peut donc compléter le tri en utilisant deux copies de TRIEUSE-BITONIQUE[$n/2$] pour trier les deux moitiés récursivement. Dans la figure 27.9(a), la récursivité est affichée explicitement, et dans la figure 27.9(b), la récursivité a été déroulée pour montrer les séparateurs, de plus en plus petits, qui constituent le reste de la trieuse bitonique. La profondeur $D(n)$ de TRIEUSE-BITONIQUE[n] est donnée par la récurrence

$$D(n) = \begin{cases} 0 & \text{si } n = 1, \\ D(n/2) + 1 & \text{si } n = 2^k \text{ et } k \geq 1, \end{cases}$$

dont la solution est $D(n) = \lg n$.

Donc, une séquence bitonique zéro-un peut être triée par TRIEUSE-BITONIQUE, dont la profondeur est $\lg n$. L'analogie du principe du zéro-un donné dans l'exercice 27.3.6 permet alors d'affirmer qu'une séquence bitonique quelconque de nombres arbitraires peut être triée par ce réseau.

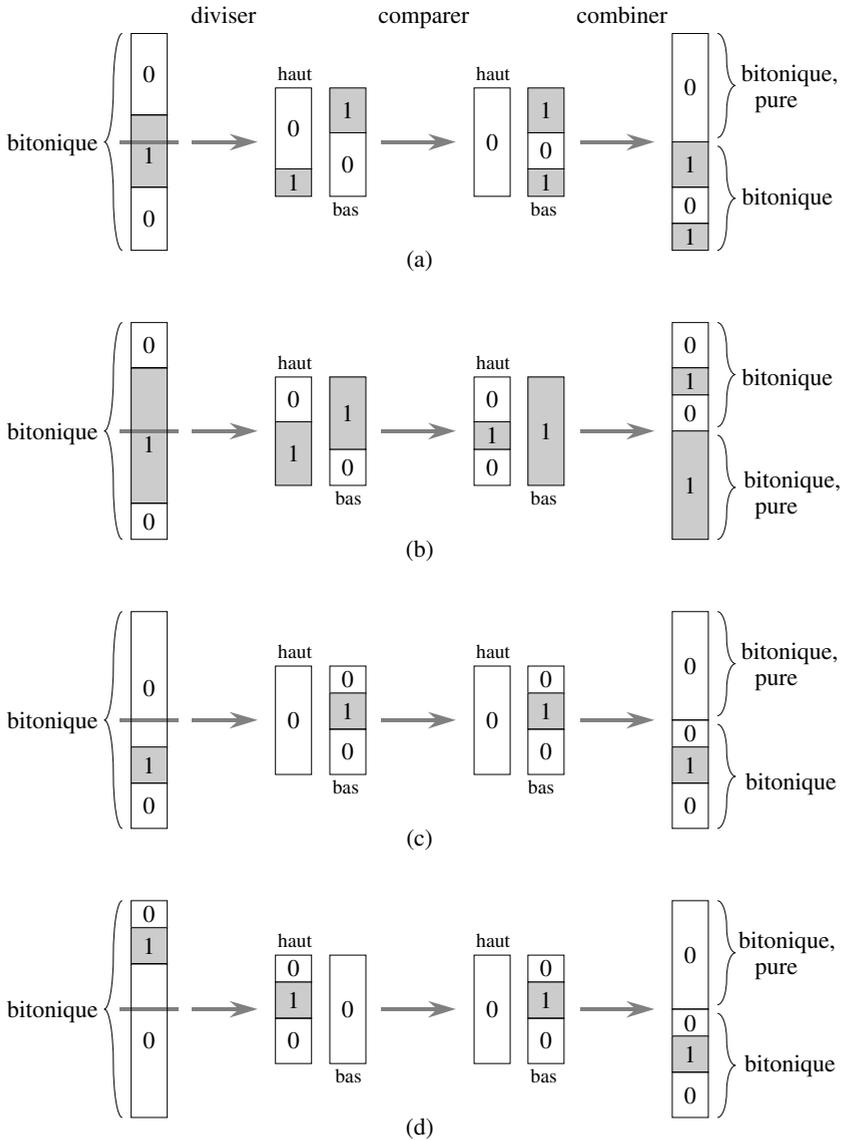


Figure 27.8 Les comparaisons possibles dans le SÉPARATEUR[n]. On suppose que la séquence d'entrée est une séquence bitonique de 0 et de 1, et qu'elle est de la forme $00 \dots 011 \dots 100 \dots 0$. Les sous-séquences de 0 apparaissent en blanc, et les sous-séquences de 1 en gris. On peut considérer que les n entrées sont séparées en deux moitiés de telle sorte que pour $i = 1, 2, \dots, n/2$, la comparaison s'effectue entre les entrées i et $i + n/2$. (a)–(b) Cas où la séparation a lieu dans la sous-séquence de 1 médiane. (c)–(d) Cas où la séparation a lieu dans une sous-séquence de 0. Pour tous les cas, tout élément de la moitié haute de la sortie est inférieur ou égal à tout élément de la moitié basse, les deux moitiés sont bitoniques et l'une au moins des moitiés est pure.

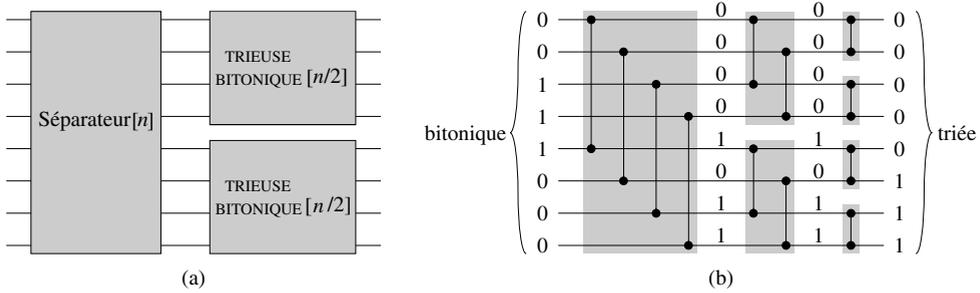


Figure 27.9 Le réseau de comparaison TRIEUSE-BITONIQUE[n], montré ici pour $n = 8$. (a) La construction récursive : SÉPARATEUR[n] suivi de deux copies de TRIEUSE-BITONIQUE[$n/2$] qui agissent en parallèle. (b) Le réseau après déroulement de la récursivité. Chaque séparateur apparaît en gris. Les câbles portent des valeurs d’essai égales à zéro ou un.

Exercices

27.3.1 Combien existe-t-il de séquences bitoniques zéro-un de longueur n ?

27.3.2 Montrer que TRIEUSE-BITONIQUE[n], où n est une puissance exacte de 2, contient $\Theta(n \lg n)$ comparateurs.

27.3.3 Décrire la manière dont on peut construire un trieuse bitonique de profondeur $O(\lg n)$ quand le nombre n d’entrées n’est pas une puissance exacte de 2.

27.3.4 Si l’entrée d’un séparateur est une séquence bitonique de nombres arbitraires, démontrer que la sortie vérifie les propriétés suivantes : les moitiés hautes et basses sont toutes les deux bitoniques, et tout élément de la moitié haute est inférieur ou égal à tout élément de la moitié basse.

27.3.5 On considère deux séquences de 0 et 1. Démontrer que, si tout élément d’une séquence est inférieur ou égal à tout élément de l’autre, alors l’une des deux séquences est pure.

27.3.6 Démontrer l’analogie suivante du principe du zéro-un pour réseaux de tri bitoniques : un réseau de comparaison qui peut trier une séquence bitonique quelconque formée de 0 et de 1 peut trier une séquence bitonique quelconque de nombres arbitraires.

27.4 UN RÉSEAU DE FUSION

Notre réseau de tri sera construit à partir de *réseaux de fusion*, qui sont des réseaux capables de fusionner deux séquences triées pour produire une séquence unique triée. On modifie TRIEUSE-BITONIQUE[n] pour créer le réseau de fusion FUSIONNEUR[n]. Comme avec la trieuse bitonique, nous n’allons démontrer la validité du réseau de fusion que pour des entrées zéro-un. L’exercice 27.4.1 demandera de montrer comment étendre la démonstration à des entrées arbitraires.

Le principe du réseau de fusion est le suivant. Étant données deux séquences triées, on obtient une nouvelle séquence bitonique en inversant l'ordre de la seconde séquence puis en concaténant les deux séquences. Par exemple, étant données les séquences zéro-un triées $X = 00000111$ et $Y = 00001111$, on inverse Y pour obtenir $Y^R = 11110000$. La concaténation de X et Y^R donne 0000011111110000 , qui est une séquence bitonique. Donc, pour fusionner deux séquences X et Y , il suffit d'effectuer un tri bitonique sur X concaténé à Y^R .

On peut construire $FUSIONNEUR[n]$ en modifiant le premier séparateur de $TRIEUSE-BITONIQUE[n]$. Le point clé est d'effectuer implicitement le renversement de la deuxième moitié de l'entrée. Étant données deux séquences $\langle a_1, a_2, \dots, a_{n/2} \rangle$ et $\langle a_{n/2+1}, a_{n/2+2}, \dots, a_n \rangle$ à fusionner, on souhaite obtenir le même effet qu'un tri bitonique sur la séquence $\langle a_1, a_2, \dots, a_{n/2}, a_n, a_{n-1}, \dots, a_{n/2+1} \rangle$. Puisque le premier séparateur de $TRIEUSE-BITONIQUE[n]$ compare les entrées i et $n/2 + i$, pour $i = 1, 2, \dots, n/2$, la première étape du réseau de fusion comparera les entrées i et $n - i + 1$. La figure 27.10 montre la correspondance. La seule subtilité est que l'ordre des sorties de la moitié inférieure de la première étape de $FUSIONNEUR[n]$ est inversé par rapport à l'ordre des sorties d'un séparateur ordinaire. Toutefois, comme le renversement d'une séquence bitonique est encore bitonique, les sorties haute et basse de la première étape du réseau de fusion vérifient les propriétés du lemme 27.3, et donc le haut et le bas peuvent être triés bitoniquement en parallèle pour produire la sortie triée du réseau de fusion.

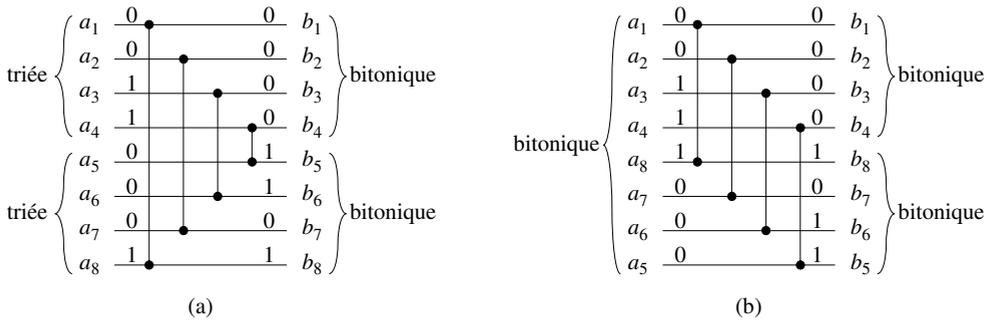


Figure 27.10 Comparaison entre la première étape de $FUSIONNEUR[n]$ et $SÉPARATEUR[n]$, pour $n = 8$. (a) La première étape de $FUSIONNEUR[n]$ transforme les deux séquences d'entrées monotones $\langle a_1, a_2, \dots, a_{n/2} \rangle$ et $\langle a_{n/2+1}, a_{n/2+2}, \dots, a_n \rangle$ en deux séquences bitoniques $\langle b_1, b_2, \dots, b_{n/2} \rangle$ et $\langle b_{n/2+1}, b_{n/2+2}, \dots, b_n \rangle$. (b) L'opération équivalente pour $SÉPARATEUR[n]$. La séquence bitonique d'entrée $\langle a_1, a_2, \dots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \dots, a_{n/2+2}, a_{n/2+1} \rangle$ est transformée en deux séquences bitoniques $\langle b_1, b_2, \dots, b_{n/2} \rangle$ et $\langle b_n, b_{n-1}, \dots, b_{n/2+1} \rangle$.

Le réseau de fusion résultant est montré à la figure 27.11. $FUSIONNEUR[n]$ et $TRIEUSE-BITONIQUE[n]$ ne diffèrent que par la première étape. La profondeur de $FUSIONNEUR[n]$ est donc $\lg n$, comme pour $TRIEUSE-BITONIQUE[n]$.

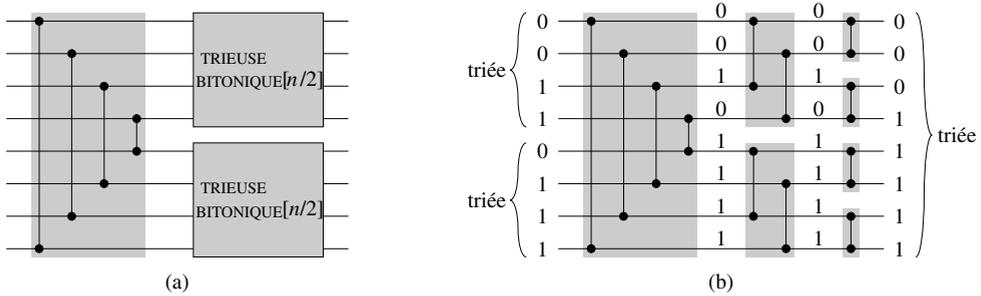


Figure 27.11 Un réseau qui fusionne deux séquences triées pour en faire une séquence triée unique. Le réseau FUSIONNEUR[n] peut être vu comme une version de TRIEUSE-BITONIQUE[n] dans laquelle le premier séparateur a été modifié pour comparer les entrées i et $n - i + 1$ pour $i = 1, 2, \dots, n/2$. Ici, $n = 8$. (a) Le réseau décomposé en deux parties : la première étape, puis les deux copies parallèles de TRIEUSE-BITONIQUE[$n/2$]. (b) Le même réseau avec la récursivité déroulée. Des valeurs d'essai apparaissent sur les câbles, et les différents paliers sont en gris.

Exercices

27.4.1 Établir une analogie du principe du zéro-un pour les réseaux de fusion. Plus précisément, montrer qu'un réseau de comparaison capable de fusionner deux séquences monotones croissantes quelconques composées de 0 et de 1 peut trier deux séquences monotones croissantes quelconques composées de nombres arbitraires.

27.4.2 Combien de séquences zéro-un différentes faut-il appliquer à l'entrée d'un réseau de comparaison pour vérifier que c'est un réseau de fusion ?

27.4.3 Montrer qu'un réseau qui peut fusionner 1 élément et $n - 1$ éléments triés pour produire une séquence triée de longueur n doit avoir une profondeur au moins égale à $\lg n$.

27.4.4 * On considère un réseau de fusion avec les entrées a_1, a_2, \dots, a_n , où n est une puissance exacte de 2, dans lequel les deux séquences monotones à fusionner sont $\langle a_1, a_3, \dots, a_{n-1} \rangle$ et $\langle a_2, a_4, \dots, a_n \rangle$. Démontrer que le nombre de comparateurs dans ce type de réseaux de fusion est $\Omega(n \lg n)$. Pourquoi est-ce un minorant intéressant ? (*Conseil* : Partitionner les comparateurs en trois ensembles.)

27.4.5 * Démontrer que tout réseau de fusion nécessite $\Omega(n \lg n)$ comparateurs, quel que soit l'ordre de ses entrées.

27.5 UN RÉSEAU DE TRI

Nous disposons à présent de tous les outils nécessaires pour construire un réseau capable de trier une séquence d'entrée quelconque. Le réseau de tri TRIEUSE[n] utilise le réseau de fusion pour implémenter une version parallèle du tri par fusion vu à la section 2.3.1. La construction et l'action du réseau de tri sont illustrées par la figure 27.12.

La figure 27.12(a) montre la construction récursive de TRIEUSE[n]. Les n éléments d'entrée sont triés en utilisant récursivement deux copies de TRIEUSE[n/2] pour trier (en parallèle) deux sous-séquences de longueur $n/2$ chacune. Les deux séquences obtenues sont ensuite fusionnées par FUSIONNEUR[n]. Le cas limite de la récursivité se produit pour $n = 1$, où l'on peut utiliser un câble pour trier la séquence à 1 élément puisqu'elle est déjà triée par nature. La figure 27.12(b) montre le résultat du déroulement de la récursivité, et la figure 27.12(c) montre le réseau réel obtenu après remplacement des boîtes FUSIONNEUR de la figure 27.12(b) par les réseaux de fusion réels.

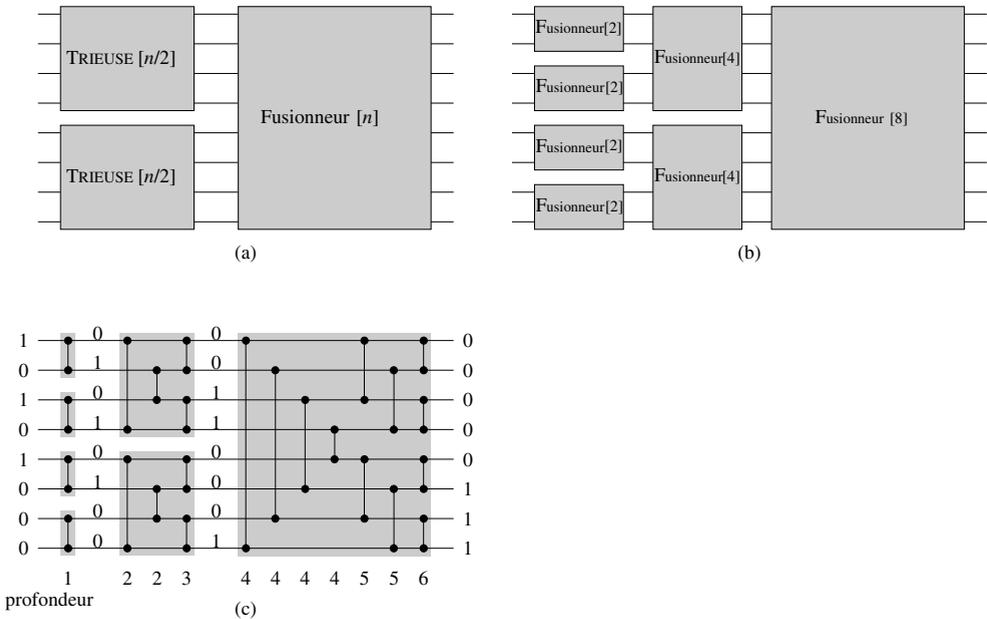


Figure 27.12 Le réseau de tri TRIEUSE[n] construit par combinaison récursive de réseaux de fusion. (a) La construction récursive. (b) Déroulement de la récursivité. (c) Remplacement des boîtes FUSIONNEUR par les réseaux de fusion réels. La profondeur de chaque comparateur est indiquée, et on peut voir des valeurs d'essai sur les câbles.

Les données passent à travers $\lg n$ paliers dans le réseau TRIEUSE[n]. Chacune des entrées du réseau est déjà une séquence de 1 élément triée. Le premier palier de TRIEUSE[n] est constitué de $n/2$ copies de FUSIONNEUR[2] qui travaillent en parallèle pour fusionner des paires de séquences de 1 élément pour produire des séquences triées de longueur 2. La seconde étape est composée de $n/4$ copies de FUSIONNEUR[4] qui fusionnent des paires de séquences triées de 2 éléments pour produire des séquences triées de longueur 4. Dans le cas général, pour $k = 1, 2, \dots, \lg n$, l'étape k est composée de $n/2^k$ copies de FUSIONNEUR[2^k] qui fusionnent des paires de séquences triées de 2^{k-1} éléments, pour produire des séquences triées de longueur

2^k . Au dernier palier, un séquence triée unique constituée de toutes les valeurs d'entrée est produite. On peut montrer par récurrence que ce réseau de tri est capable de trier des séquences zéro-un ; donc, d'après le principe du zéro-un (théorème 27.2), il peut trier des valeurs arbitraires.

La profondeur du réseau de tri peut s'analyser récursivement. La profondeur $D(n)$ de $\text{TRIEUSE}[n]$ est égale à la profondeur $D(n/2)$ de $\text{TRIEUSE}[n/2]$ (il existe deux copies de $\text{TRIEUSE}[n/2]$, mais elles agissent en parallèle) plus la profondeur $\lg n$ de $\text{FUSIONNEUR}[n]$. Donc, la profondeur de $\text{TRIEUSE}[n]$ est donnée par la récurrence

$$D(n) = \begin{cases} 0 & \text{si } n = 1, \\ D(n/2) + \lg n & \text{si } n = 2^k \text{ et } k \geq 1, \end{cases}$$

dont la solution est $D(n) = \Theta(\lg^2 n)$. (Utiliser la version de la méthode générale donnée à l'exercice 4.4.4). On peut donc trier n nombres en parallèle avec un temps $O(\lg^2 n)$.

Exercices

27.5.1 Combien y a-t-il de comparateurs dans $\text{TRIEUSE}[n]$?

27.5.2 Montrer que la profondeur de $\text{TRIEUSE}[n]$ vaut exactement $(\lg n)(\lg n + 1)/2$.

27.5.3 On dispose de $2n$ éléments $\langle a_1, a_2, \dots, a_{2n} \rangle$ qu'on souhaite partitionner en deux sous-ensembles : les n plus petits et les n plus grands. Démontrer que cela peut se faire avec une profondeur supplémentaire constante, après tris séparés de $\langle a_1, a_2, \dots, a_n \rangle$ et de $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$.

27.5.4 * Soient $S(k)$ la profondeur d'un réseau de tri à k entrées et $M(k)$ la profondeur d'un réseaux de fusion à $2k$ entrées. On suppose qu'on a une séquence de n nombres à trier, dans laquelle chaque nombre se trouve, dans le pire des cas, à k positions de la place qu'il occuperait si la séquence était triée. Montrer qu'il est possible de trier ces n nombres avec une profondeur $S(k) + 2M(k)$.

27.5.5 * On peut trier les éléments d'une matrice $m \times m$ en répétant k fois la procédure suivante :

- 1) Trier chaque ligne à numéro impair dans un ordre monotone croissant.
- 2) Trier chaque ligne à numéro pair dans un ordre monotone décroissant.
- 3) Trier chaque colonne dans un ordre monotone croissant.

Combien d'itérations k de cette procédure faut-il effectuer pour que la matrice soit triée, et dans quel ordre doit-on lire les éléments de la matrice après les k itérations pour obtenir le sortie triée ?

PROBLÈMES

27.1. Réseaux de tri de transposition

Un réseau de comparaison est un *réseau de transposition* si chaque comparateur relie des lignes adjacentes, comme dans le réseau de la figure 27.3.

- a. Montrer qu'un réseau de tri de transposition à n entrées possède $\Omega(n^2)$ comparateurs.
- b. Démontrer qu'un réseau de transposition à n entrées est un réseau de tri si et seulement si il est capable de trier la séquence $\langle n, n - 1, \dots, 1 \rangle$. (*conseil* : Utiliser une démonstration par récurrence analogue à celle utilisée dans la démonstration du lemme 27.1.)

Un *réseau de tri pair-impair* à n entrées $\langle a_1, a_2, \dots, a_n \rangle$ est un réseau de tri de transposition à n niveaux de comparateurs reliés selon le modèle « mur de briques » de la figure 27.13. Comme on peut le voir sur cette figure, pour $i = 1, 2, \dots, n$ et $d = 1, 2, \dots, n$, la ligne i est reliée par un comparateur de profondeur d à la ligne $j = i + (-1)^{i+d}$ si $1 \leq j \leq n$.

- c. Démontrer qu'un réseau de tri pair-impair est effectivement capable de trier.

27.2. Réseau de fusion pair-impair de Batcher

À la section 27.4, nous avons vu comment construire un réseau de fusion basé sur le tri bitonique. Dans ce problème, nous allons construire un *réseau de fusion pair-impair*. On suppose que n est une puissance exacte de 2, et on souhaite fusionner la séquence triée d'éléments des lignes $\langle a_1, a_2, \dots, a_n \rangle$ et celle des lignes $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$. Si $n = 1$, on place un comparateur entre les lignes a_1 et a_2 . Sinon, on construit récursivement deux réseaux de fusion pair-impair fonctionnant en parallèle. Le premier fusionne la séquence des lignes $\langle a_1, a_3, \dots, a_{n-1} \rangle$ et celle des lignes $\langle a_{n+1}, a_{n+3}, \dots, a_{2n-1} \rangle$ (éléments impairs). Le second fusionne $\langle a_2, a_4, \dots, a_n \rangle$ et $\langle a_{n+2}, a_{n+4}, \dots, a_{2n} \rangle$ (éléments pairs). Pour combiner les deux sous-séquences triées, on place un comparateur entre a_{2i} et a_{2i+1} pour $i = 1, 2, \dots, n - 1$.

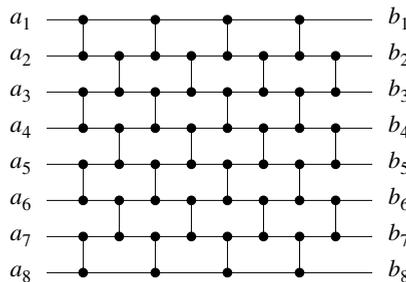


Figure 27.13 Un réseau de tri pair-impair à 8 entrées.

- Dessiner un réseau de fusion à $2n$ entrées pour $n = 4$.
- Le professeur Nimbus suggère la méthode suivante : pour combiner les deux sous-séquences triées produites par la fusion récursive, au lieu de placer un comparateur entre a_{2i} et a_{2i+1} pour $i = 1, 2, \dots, n - 1$, on place un comparateur entre a_{2i-1} et a_{2i} pour $i = 1, 2, \dots, n$. Dessiner un tel réseau à $2n$ entrées pour $n = 4$, et donner un contre-exemple pour montrer que le professeur se trompe quand il pense que le réseau ainsi fabriqué est un réseau de fusion. Montrer que le réseau de fusion à $2n$ entrées de la partie (a) fonctionne correctement sur votre exemple.
- Utiliser le principe du zéro-un pour démontrer qu'un réseau de fusion pair-impair à $2n$ entrées est bien un réseau de fusion.
- Quelle est la profondeur d'un réseau de fusion pair-impair à $2n$ entrées ? Quelle est sa taille ?

27.3. Réseaux de permutation

Un *réseau de permutation* à n entrées et n sorties possède des bascules lui permettant de relier ses entrées à ses sorties selon n'importe laquelle des $n!$ permutations possibles. La figure 27.14(a) montre le réseau de permutation P_2 à 2 entrées et 2 sorties, comportant une seule bascule pouvant alimenter les sorties soit avec les entrées telles qu'elles se présentent, soit en les croisant.

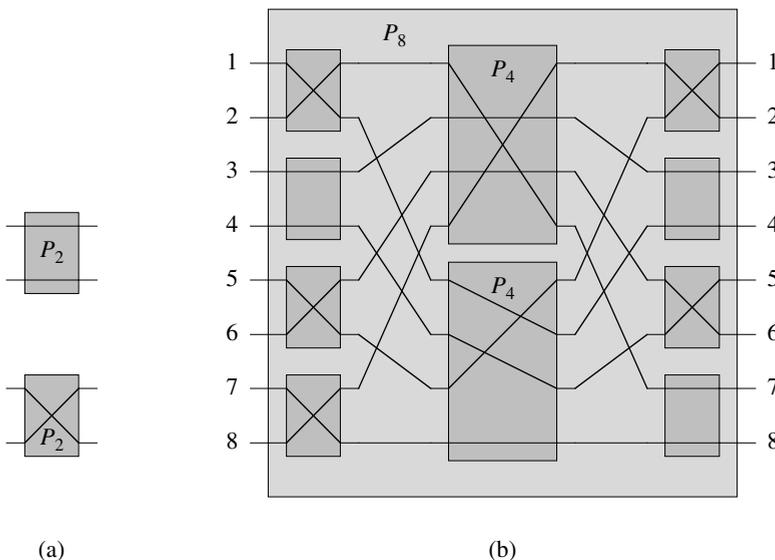


Figure 27.14 Réseaux de permutation. (a) Le réseau de permutation P_2 , composé d'une seule bascule pouvant être configurée de l'une des deux façons affichées. (b) La construction récursive de P_8 à partir de 8 bascules et deux P_4 . Les bascules et les P_4 sont configurés pour réaliser la permutation $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$.

- a. Montrer que, si l'on remplace chaque comparateur d'un réseau de tri par la bascule de la figure 27.14(a), le nouveau réseau est un réseau de permutation. Autrement dit, pour toute permutation π , il existe une façon de configurer les bascules dans le réseau telle que l'entrée i soit reliée à la sortie $\pi(i)$.

La figure 27.14(b) montre la construction récursive d'un réseau de permutation P_8 à 8 entrées et 8 sorties, qui utilise deux copies de P_4 et 8 bascules. Les bascules ont été configurées pour réaliser la permutation $\pi = \langle \pi(1), \pi(2), \dots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$, ce qui exige (récursivement) que le P_4 du haut fasse la permutation $\langle 4, 2, 3, 1 \rangle$ et que le P_4 du bas fasse la permutation $\langle 2, 3, 1, 4 \rangle$.

- b. Montrer comment réaliser la permutation $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$ sur P_8 en dessinant les configurations des bascules et les permutations effectuées par les deux P_4 .

Soit n une puissance exacte de 2. On définit P_n récursivement en fonction de deux $P_{n/2}$ de la même façon que l'on a défini P_8 .

- c. Décrire un algorithme en $O(n)$ (sur une machine à accès aléatoire ordinaire) qui configure les n bascules reliées aux entrées et aux sorties de P_n et qui spécifie les permutations à réaliser par chaque $P_{n/2}$, de façon à effectuer une permutation quelconque sur n éléments. Prouver la validité de votre algorithme.
- d. Quelles sont la profondeur et la taille de P_n ? Combien de temps faut-il sur une machine ordinaire à accès aléatoire pour calculer toutes les configurations de bascule, y compris celles situées dans les $P_{n/2}$?
- e. Montrer que, pour $n > 2$, tout réseau de permutation, pas uniquement P_n , doit réaliser une certaine permutation en utilisant deux combinaisons distinctes de configurations de bascule.

NOTES

Knuth [185] contient une étude des réseaux de tri et en dresse un historique. Ils ont été apparemment étudiés en 1954 par P. N. Armstrong, R. J. Nelson et D. J. O'Connor. Au début des années 1960, K. E. Batcher découvrit le premier réseau capable de fusionner deux séquences de n nombres en temps $O(\lg n)$. Il utilisa la fusion pair-impair (voir problème 27.2) et montra également comment utiliser cette technique pour trier n nombres en temps $O(\lg^2 n)$. Peu après, il découvrit une trieuse bitonique de profondeur $O(\lg n)$ similaire à celle présentée à la section 27.3. Knuth attribue le principe du zéro-un à W. G. Bouricius (1954), qui le démontra dans le contexte des arbres de décision.

Pendant longtemps, on se demanda si un réseau de tri pouvait avoir une profondeur $O(\lg n)$. En 1983, on trouve une réponse peu satisfaisante. Le réseau de tri AKS (du nom de ses inventeurs Ajtai, Komlós et Szemerédi [11]) est capable de trier n nombres avec une profondeur $O(\lg n)$ utilisant $O(n \lg n)$ comparateurs. Malheureusement, les constantes cachées dans la notation O sont plutôt grandes (plusieurs milliers), ce qui rend ce réseau peu pratique à utiliser.

Chapitre 28

Calcul matriciel

Les opérations sur les matrices sont au cœur du calcul scientifique. Disposer d'algorithmes efficaces pour les matrices est donc d'une importance pratique considérable. Ce chapitre propose une brève introduction à la théorie des matrices et aux opérations sur les matrices, en insistant sur les problèmes de la multiplication des matrices et de la résolution d'ensembles d'équations linéaires.

Après que la section 28.1 aura présenté les concepts et notations élémentaires ayant trait aux matrices, la section 28.2 présentera l'étonnant algorithme de Strassen, qui multiplie deux matrices $n \times n$ matrices en temps $\Theta(n^{\lg 7}) = O(n^{2,81})$. La section 28.3 montrera comment résoudre un ensemble d'équations linéaires à l'aide des décompositions LUP. La section 28.4 étudiera ensuite les similitudes entre le problème de la multiplication des matrices et le problème de l'inversion d'une matrice. Enfin, la section 28.5 se penchera sur la classe importante que sont les matrices définies positives et montrera comment elles permettent de trouver une solution de moindres carrés pour un ensemble d'équations linéaire surdéterminé.

Un problème majeur qui se pose en pratique est la *stabilité numérique*. Compte tenu de la précision limitée des représentations en virgule flottante dans les ordinateurs, il peut y avoir amplification des erreurs d'arrondi pendant le déroulement d'un calcul et donc risque de résultats faux ; de tels calculs sont numériquement instables. Nous ne traiterons qu'occasionnellement de la stabilité numérique dans ce chapitre, renvoyant le lecteur à l'excellent ouvrage de Golub et Van Loan [125] pour une étude détaillée des problèmes de stabilité.

28.1 PROPRIÉTÉS DES MATRICES

Dans cette section, nous passons en revue quelques concepts de base de la théorie des matrices, ainsi que certaines propriétés fondamentales, en insistant sur celles qui seront nécessaires dans les sections suivantes.

a) Matrices et vecteurs

Une **matrice** est un tableau de nombres rectangulaire. Par exemple,

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \end{aligned} \quad (28.1)$$

est une matrice $A = (a_{ij})$ de taille 2×3 , où pour $i = 1, 2$ et $j = 1, 2, 3$, a_{ij} est l'élément situé à la ligne i et à la colonne j . On utilise des lettres majuscules pour représenter les matrices, et les lettres minuscules correspondantes pour représenter leurs éléments. L'ensemble de toutes les matrices $m \times n$ dont les éléments ont des valeurs réelles est noté $\mathbf{R}^{m \times n}$. D'une manière générale, l'ensemble des matrices $m \times n$ dont les éléments sont pris dans l'ensemble S est noté $S^{m \times n}$.

La **transposée** d'une matrice A est la matrice ${}^T A$ obtenue en échangeant les lignes et les colonnes de A . Pour la matrice A de l'équation (28.1),

$${}^T A = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Un **vecteur** est un tableau de nombres à une dimension. Par exemple,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \quad (28.2)$$

est un vecteur de taille 3. On utilise des lettres minuscules pour représenter les vecteurs, et la i ème élément d'un vecteur x de taille n est notée x_i , pour $i = 1, 2, \dots, n$. On considère que la forme standard d'un vecteur est un **vecteur colonne**, équivalent à une matrice $n \times 1$; le **vecteur ligne** correspondant est obtenu en prenant la transposée :

$${}^T x = (2 \ 3 \ 5).$$

Le **vecteur unité** e_i est le vecteur dont le i ème élément est égal à 1 et tous les autres éléments sont égaux à 0. Le plus souvent, la taille d'un vecteur unité est donnée par le contexte.

Une **matrice nulle** est une matrice dont tous les éléments sont égaux à 0. On notera souvent 0 une telle matrice, l'ambiguïté entre le nombre 0 et une matrice de 0 étant facile à résoudre en fonction du contexte. S'il est question d'une matrice de 0, sa taille devra également être déduite du contexte.

On rencontre fréquemment des matrices **carrées** $n \times n$. Quelques cas particuliers de matrices carrées sont particulièrement intéressants :

- 1) Dans une **matrice diagonale**, on a $a_{ij} = 0$ quand $i \neq j$. Comme tous les éléments situés en dehors de la diagonale valent zéro, la matrice peut être déterminée par l'énoncé des éléments situés sur la diagonale :

$$\text{diag}(a_{11}, a_{22}, \dots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix}.$$

- 2) La **matrice identité** I_n est une matrice diagonale $n \times n$ avec des 1 sur la diagonale :

$$\begin{aligned} I_n &= \text{diag}(1, 1, \dots, 1) \\ &= \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}. \end{aligned}$$

Lorsque I apparaît sans indice, on peut déduire sa taille du contexte. La i ème colonne d'une matrice identité est le vecteur unité e_i .

- 3) Dans une **matrice tridiagonale** T , on a $t_{ij} = 0$ si $|i - j| > 1$. Les éléments non nuls n'apparaissent que sur la diagonale principale, juste au-dessus de la diagonale principale ($t_{i,i+1}$ pour $i = 1, 2, \dots, n - 1$), ou juste en dessous de la diagonale principale ($t_{i+1,i}$ pour $i = 1, 2, \dots, n - 1$) :

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \dots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \dots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \dots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \dots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

- 4) On appelle **matrice triangulaire supérieure** une matrice U pour laquelle $u_{ij} = 0$ si $i > j$. Tous les éléments situés sous la diagonale sont égaux à zéro :

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}.$$

Une matrice triangulaire supérieure est **unitaire** si sa diagonale est entièrement composée de 1.

- 5) On appelle **matrice triangulaire inférieure** une matrice L pour laquelle $l_{ij} = 0$ si $i < j$. Tous les éléments situés au-dessus de la diagonale sont égaux à zéro :

$$L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}.$$

Une matrice triangulaire inférieure est **unitaire** si sa diagonale est entièrement composée de 1.

- 6) Une **matrice de permutation** P comporte exactement un 1 dans chaque ligne ou colonne, et 0 partout ailleurs. Voici un exemple de matrice de permutation :

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Une telle matrice est appelée matrice de permutation parce que la multiplication d'un vecteur x par une matrice de permutation a pour effet de permuter (réordonner) les éléments de x .

- 7) Une **matrice symétrique** A satisfait la condition $A = {}^T A$. Par exemple,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

est une matrice symétrique.

b) Opérations sur les matrices

Les éléments d'une matrice ou d'un vecteur sont des nombres pris dans un système numérique particulier, tel les nombres réels, les nombres complexes ou les entiers modulo un certain nombre premier. Le système numérique définit la manière d'additionner et de multiplier les nombres. On peut étendre ces définitions pour y englober l'addition et la multiplication des matrices.

On définit l'**addition de matrices** de la manière suivante. Si $A = (a_{ij})$ et $B = (b_{ij})$ sont des matrices $m \times n$, alors leur somme $C = (c_{ij}) = A + B$ est la matrice $m \times n$ définie par

$$c_{ij} = a_{ij} + b_{ij}$$

pour $i = 1, 2, \dots, m$ et $j = 1, 2, \dots, n$. Autrement dit, l'addition des matrices s'effectue au niveau des éléments. La matrice nulle est l'identité pour l'addition des

matrices :

$$\begin{aligned} A + 0 &= A \\ &= 0 + A . \end{aligned}$$

Si λ est un nombre et $A = (a_{ij})$ une matrice, alors $\lambda A = (\lambda a_{ij})$ est le **produit scalaire** de A obtenu en multipliant chacune de ses éléments par λ . En particulier, on définit l'**opposé** d'une matrice $A = (a_{ij})$ par $-1 \cdot A = -A$, de sorte que la ij ème élément de $-A$ prend la valeur $-a_{ij}$. Donc,

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A . \end{aligned}$$

Étant donné cette définition, on peut définir la **soustraction de matrices** comme l'addition de l'opposé d'une matrice : $A - B = A + (-B)$.

La **multiplication de matrices** est définie de la manière suivante. On commence avec deux matrices A et B **compatibles**, au sens où le nombre de colonnes de A est égal au nombre de lignes de B . (D'une façon générale, on supposera toujours, dans une expression contenant un produit de matrices AB , que les matrices A et B sont compatibles.) Si $A = (a_{ij})$ est une matrice $m \times n$ et $B = (b_{jk})$ une matrice $n \times p$, leur produit $C = AB$ est la matrice $C = (c_{ik})$, de taille $m \times p$, où

$$c_{ik} = \sum_{j=1}^n a_{ij} b_{jk} \quad (28.3)$$

pour $i = 1, 2, \dots, m$ et $k = 1, 2, \dots, p$. La procédure MULTIPLIER-MATRICES de la section 25.1 implémente la multiplication des matrices en adaptant directement l'équation (28.3) et en supposant que les matrices sont carrées : $m = n = p$. Pour multiplier des matrices $n \times n$, MULTIPLIER-MATRICES effectue n^3 multiplications et $n^2(n - 1)$ additions, de sorte que son temps d'exécution est $\Theta(n^3)$.

Les matrices possèdent une bonne partie (mais pas toutes) des propriétés algébriques classiques des nombres. La matrice identité est l'identité pour la multiplication :

$$I_m A = A I_n = A$$

pour une matrice $m \times n$ quelconque A . La multiplication par la matrice nulle donne une matrice nulle :

$$A 0 = 0 .$$

La multiplication des matrices est associative :

$$A(BC) = (AB)C \quad (28.4)$$

pour des matrices compatibles A , B et C . La multiplication des matrices est distributive par rapport à l'addition :

$$\begin{aligned} A(B + C) &= AB + AC , \\ (B + C)D &= BD + CD . \end{aligned} \quad (28.5)$$

La multiplication de matrices $n \times n$ n'est pas commutative, sauf si $n = 1$. Par exemple, si $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ et $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, alors

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

et

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Les produits matrice-vecteur ou vecteur-vecteur sont définis comme si le vecteur était la matrice $n \times 1$ équivalente (ou une matrice $1 \times n$, dans le cas d'un vecteur ligne). Donc, si A est une matrice $m \times n$ et x un vecteur de taille n , alors Ax est un vecteur de taille m . Si x et y sont des vecteurs de taille n , alors

$${}^Txy = \sum_{i=1}^n x_i y_i$$

est un nombre (une matrice 1×1 en réalité) appelé **produit scalaire** de x et y . La matrice $x^T y$ est une matrice Z de taille $n \times n$, appelée **produit extérieur** de x et y , avec $z_{ij} = x_i y_j$. La **norme (euclidienne)** $\|x\|$ d'un vecteur x de taille n est définie par

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2} \\ &= ({}^Txx)^{1/2}. \end{aligned}$$

Donc, la norme de x est égale à sa longueur dans l'espace euclidien à n dimensions.

c) Inversions de matrice, rangs et déterminants

On définit l'**inverse** d'une matrice A de taille $n \times n$ comme étant une matrice $n \times n$, notée A^{-1} (si elle existe), telle que $AA^{-1} = I_n = A^{-1}A$. Par exemple,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

De nombreuses matrices $n \times n$ n'ont pas d'inverse, bien que n'étant pas nulles. Une matrice sans inverse est dite **non inversible**, ou **singulière**. Voici un exemple de matrice non nulle singulière :

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Si une matrice possède un inverse, elle est dite **inversible**, ou **non singulière**. Les inverses de matrice, quand ils existent, sont uniques. (Voir exercice 28.1.3.) Si A et B sont deux matrices $n \times n$ non singulières, alors

$$(BA)^{-1} = A^{-1}B^{-1}. \quad (28.6)$$

L'opération inversion est commutative avec l'opération transposition :

$${}^T(A^{-1}) = ({}^T A)^{-1} .$$

Les vecteurs x_1, x_2, \dots, x_n sont **linéairement dépendants** s'il existe des coefficients c_1, c_2, \dots, c_n non tous nuls, tels que $c_1x_1 + c_2x_2 + \dots + c_nx_n = 0$. Par exemple, les vecteurs ligne $x_1 = {}^T(1 \ 2 \ 3)$, $x_2 = {}^T(2 \ 6 \ 4)$ et $x_3 = {}^T(4 \ 11 \ 9)$ sont linéairement dépendants, puisque $2x_1 + 3x_2 - 2x_3 = 0$. Si des vecteurs ne sont pas linéairement dépendants, ils sont dits **linéairement indépendants**. Par exemple, les colonnes d'une matrice identité sont linéairement indépendantes.

Le **rang colonne** d'une matrice A non nulle $m \times n$ est la taille du plus grand ensemble linéairement indépendant de colonnes de A . De même, le **rang ligne** de A est la taille du plus grand ensemble linéairement indépendant de lignes de A . Une propriété fondamentale d'une matrice A quelconque est que son rang ligne est toujours égal à son rang colonne ; il suffit donc de parler de **rang** de A . Le rang d'une matrice $m \times n$ est un entier compris entre 0 et $\min(m, n)$, inclus. (Le rang d'une matrice nulle vaut 0, et celui d'une matrice identité $n \times n$ est égal à n .) Une définition équivalente, mais souvent plus utile, est que le rang d'une matrice A non nulle $m \times n$ est le plus petit nombre r tel qu'il existe deux matrices B et C de tailles respectives $m \times r$ et $r \times n$ telles que

$$A = BC .$$

Une matrice carrée $n \times n$ a un **rang plein** si son rang est égal à n . Une matrice $m \times n$ a un **rang colonne plein** si son rang vaut n . Le théorème ci-après donne une propriété fondamentale des rangs.

Théorème 28.1 *Une matrice carrée a un rang plein si et seulement si elle est non singulière.*

Une matrice $m \times n$ a un **rang colonne plein** si son rang est égal à n .

Un **vecteur d'annulation** d'une matrice A est un vecteur non nul x tel que $Ax = 0$. Le théorème suivant, dont la démonstration est laissée en exercice (voir exercice 28.1.9), et son corollaire, relie les notions de rang colonne et de singularité aux vecteurs d'annulation.

Théorème 28.2 *Une matrice A a un rang colonne plein si et seulement si elle ne possède pas de vecteur d'annulation.*

Corollaire 28.3 *Une matrice carrée A est singulière si et seulement si elle possède un vecteur d'annulation.*

Le i jème **mineur** d'une matrice A de taille $n \times n$, pour $n > 1$, est la matrice $A_{[ij]}$ de taille $(n - 1) \times (n - 1)$ obtenue en supprimant la i ème ligne et la j ème colonne de

A. Le **déterminant** d'une matrice A de taille $n \times n$ peut être défini récursivement en fonction de ses mineurs par :

$$\det(A) = \begin{cases} a_{11} & \text{si } n = 1, \\ \sum_{j=1}^n (-1)^{1+j} a_{1j} \det(A_{[1j]}) & \text{si } n > 1. \end{cases} \quad (28.7)$$

Le terme $(-1)^{i+j} \det(A_{[ij]})$ s'appelle le **cofacteur** de l'élément a_{ij} .

Les théorèmes suivants, dont les démonstrations sont omises, donnent les propriétés fondamentales des déterminants.

Théorème 28.4 (Propriétés du déterminant) *Le déterminant d'une matrice carrée A possède les propriétés suivantes :*

- Si une ligne ou une colonne quelconque de A est égale à zéro, alors $\det(A) = 0$.
- Le déterminant de A est multiplié par λ si les éléments d'une ligne (ou colonne) quelconque de A sont tous multipliés par λ .
- Le déterminant de A reste inchangé si les éléments d'une ligne (resp. colonne) sont ajoutées à ceux d'une autre ligne (resp. colonne).
- Le déterminant de A est égal au déterminant de ${}^T A$.
- Le déterminant de A est multiplié par -1 si deux lignes (resp. colonnes) sont échangées.

D'autre part, pour deux matrices carrées A et B quelconques, on a

$$\det(AB) = \det(A) \det(B).$$

Théorème 28.5 *Une matrice A de taille $n \times n$ est singulière si et seulement si $\det(A) = 0$.*

d) Matrices définies positives

Les matrices définies positives jouent un rôle important dans de nombreuses applications. Une matrice A de taille $n \times n$ est **définie positive** si ${}^T x A x > 0$ pour tout vecteur $x \neq 0$ de taille n . Par exemple, la matrice identité est définie positive, puisque pour un vecteur non nul quelconque, $x = {}^T (x_1 \ x_2 \ \cdots \ x_n)$,

$$\begin{aligned} {}^T x I_n x &= {}^T x x \\ &= \|x\|^2 \\ &= \sum_{i=1}^n x_i^2 \\ &> 0. \end{aligned}$$

Comme on le verra, les matrices apparaissant dans les applications sont souvent définies positives en raison du théorème suivant.

Théorème 28.6 Pour une matrice A quelconque de rang colonne plein, la matrice TAA est définie positive.

Démonstration : Il faut montrer que ${}^Tx({}^TAA)x > 0$ pour tout vecteur x non nul. Pour un vecteur x quelconque,

$$\begin{aligned} {}^Tx({}^TAA)x &= {}^T(Ax)(Ax) && \text{(d'après l'exercice 28.1.2)} \\ &= \|Ax\|^2 \\ &\geq 0. \end{aligned}$$

Notez que $\|Ax\|^2$ est justement égal à la somme des carrés des éléments du vecteur Ax . Donc, $\|Ax\|^2 \geq 0$. Si $\|Ax\|^2 = 0$, les éléments de Ax sont tous égaux à 0, ce qui revient à dire que $Ax = 0$. Comme A a un rang colonne plein, $Ax = 0$ implique $x = 0$ d'après le théorème 28.2. On en déduit que TAA est définie positive. \square

D'autres propriétés des matrices définies positives seront explorées à la section 28.5.

Exercices

28.1.1 Montrer que, si A et B sont des matrices symétriques $n \times n$, il en est de même de $A + B$ et $A - B$.

28.1.2 Prouver que $(AB)^T = B^T A^T$ et que $A^T A$ est toujours une matrice symétrique.

28.1.3 Prouver que les inverses de matrice sont uniques ; en d'autres termes, si B et C sont des inverses de A , alors $B = C$.

28.1.4 Démontrer que le produit de deux matrices triangulaires inférieures est une matrice triangulaire inférieure. Démontrer que le déterminant d'une matrice triangulaire (inférieure ou supérieure) est égal au produit de ses éléments diagonaux. Démontrer que l'inverse d'une matrice triangulaire inférieure, s'il existe, est une matrice triangulaire inférieure.

28.1.5 Démontrer que si P est une matrice de permutation $n \times n$ et A une matrice $n \times n$, alors PA peut être obtenue à partir de A en permutant ses lignes, et AP peut être obtenue à partir de A en permutant ses colonnes. Démontrer que le produit de deux matrices de permutation est une matrice de permutation. Démontrer que si P est une matrice de permutation, alors P est inversible, son inverse est ${}^T P$, et ${}^T P$ est une matrice de permutation.

28.1.6 Soit A et B deux matrices $n \times n$ telles que $AB = I$. Démontrer que si A' est obtenue à partir de A en ajoutant la ligne j à la ligne i , alors l'inverse B' de A' peut être obtenue par soustraction de la colonne i à la colonne j de B .

28.1.7 Soit A une matrice non singulière $n \times n$ à éléments complexes. Montrer que tout élément de A^{-1} est réel si et seulement si tout élément de A est réel.

28.1.8 Montrer que si A est une matrice symétrique non singulière $n \times n$, alors A^{-1} est symétrique. Montrer que si B est une matrice arbitraire $m \times n$, alors la matrice $m \times m$ donnée par le produit ${}^T BAB$ est symétrique.

28.1.9 Prouver le théorème 28.2. C'est à dire, montrer qu'une matrice A a un rang colonne plein si et seulement si $Ax = 0$ implique $x = 0$. (*Conseil* : Exprimer la dépendance linéaire d'une colonne par rapport aux autres sous la forme d'une équation matrice-vecteur.)

28.1.10 Démontrer que pour deux matrices compatibles A et B quelconques,

$$\text{rang}(AB) \leq \min(\text{rang}(A), \text{rang}(B)),$$

où l'on a égalité si A ou B est une matrice carrée non singulière. (*Conseil* : Utiliser l'autre définition du rang d'une matrice.)

28.1.11 Étant donnés les nombres x_0, x_1, \dots, x_{n-1} , démontrer que le déterminant de la *matrice de Vandermonde*

$$V(x_0, x_1, \dots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

est

$$\det(V(x_0, x_1, \dots, x_{n-1})) = \prod_{0 \leq j < k \leq n-1} (x_k - x_j).$$

(*conseil* : Multiplier la colonne i par $-x_0$ et l'ajouter à la colonne $i+1$ pour $i = n-1, n-2, \dots, 1$, puis utiliser une récurrence.)

28.2 ALGORITHME DE STRASSEN POUR LA MULTIPLICATION DES MATRICES

Cette section présente le remarquable algorithme récursif de Strassen, qui permet de multiplier des matrices $n \times n$ en temps $\Theta(n^{\lg 7}) = O(n^{2,81})$. Pour des valeurs assez grandes de n , il est donc meilleur que l'algorithme naïf à temps $\Theta(n^3)$ MULTIPLIER-MATRICES vu à la section 25.1.

a) Aperçu de l'algorithme

L'algorithme de Strassen peut être vu comme une application de la bonne vieille technique diviser-pour-régner. Supposons qu'on veuille calculer le produit $C = AB$, où A, B et C sont des matrices $n \times n$. En supposant que n soit une puissance exacte de 2, on divise A, B et C chacune en quatre matrices $n/2 \times n/2$, ce qui fait réécrire l'équation $C = AB$ de la manière suivante :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}. \quad (28.8)$$

(L'exercice 28.2.2 s'intéresse à la situation où n n'est pas une puissance exacte de 2.)
L'équation (28.8) correspond aux quatre équations

$$r = ae + bg, \quad (28.9)$$

$$s = af + bh, \quad (28.10)$$

$$t = ce + dg, \quad (28.11)$$

$$u = cf + dh. \quad (28.12)$$

Chacune de ces quatre équations spécifie deux multiplications de matrices $n/2 \times n/2$ et l'addition de leurs produits $n/2 \times n/2$. Ces équations permettent de définir une stratégie diviser-pour-régner immédiate, qui donne la récurrence suivante pour le temps $T(n)$ requis pour multiplier deux matrices $n \times n$:

$$T(n) = 8T(n/2) + \Theta(n^2). \quad (28.13)$$

Malheureusement, la récurrence (28.13) a pour solution $T(n) = \Theta(n^3)$, et cette méthode n'est donc pas plus rapide que la méthode ordinaire.

Strassen a découvert une approche récursive différente, qui ne demande que 7 multiplications récursives de matrices $n/2 \times n/2$, et $\Theta(n^2)$ additions et soustractions scalaires, ce qui aboutit à la récurrence

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \\ &= \Theta(n^{\lg 7}) \\ &= O(n^{2,81}). \end{aligned} \quad (28.14)$$

La méthode de Strassen est composée de quatre étapes :

- 1) Diviser les matrices d'entrée A et B en sous-matrices $n/2 \times n/2$, comme dans l'équation (28.8).
- 2) A l'aide de $\Theta(n^2)$ additions et soustractions scalaires, calculer 14 matrices $A_1, B_1, A_2, B_2, \dots, A_7, B_7$ de dimension $n/2 \times n/2$.
- 3) Calculer récursivement les sept produits de matrices $P_i = A_i B_i$ pour $i = 1, 2, \dots, 7$.
- 4) Calculer les sous-matrices désirées r, s, t, u de la matrice résultat C en additionnant et/ou soustrayant diverses combinaisons des matrices P_i , à l'aide de $\Theta(n^2)$ additions et soustractions scalaires seulement.

Cette procédure vérifie la récurrence (28.14). Il ne reste donc plus qu'à expliciter les détails.

b) Détermination des produits de sous-matrices

On ne sait pas exactement comment Strassen a découvert les produits de sous-matrices qui constituent la partie fondamentale de son algorithme. Nous allons ici reconstruire une méthode de découverte plausible.

Nous supposons que chaque produit de matrice P_i peut s'écrire sous la forme

$$\begin{aligned} P_i &= A_i B_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h), \end{aligned} \quad (28.15)$$

où les coefficients α_{ij}, β_{ij} sont tous pris dans l'ensemble $\{-1, 0, 1\}$. Autrement dit, on conjecture que chaque produit est calculé en additionnant ou soustrayant certaines des sous-matrices de A , puis en additionnant ou soustrayant certaines des sous-matrices de B , puis en faisant le produit des deux résultats. Bien qu'il soit possible d'appliquer des stratégies plus générales, celle-ci, quoique simple, a le mérite de fonctionner.

Si tous les produits sont formés de cette manière, on peut alors utiliser cette méthode récursivement sans avoir à supposer que la multiplication est commutative, puisque chaque produit comporte toutes les sous-matrices de A à gauche et toutes les sous-matrices de B à droite. Cette propriété est essentielle pour l'application récursive de cette méthode, car la multiplication des matrices n'est pas commutative.

Par commodité, on utilisera des matrices 4×4 pour représenter les combinaisons linéaires des produits de sous-matrices, où chaque produit combine une sous-matrice de A avec une sous-matrice de B comme dans l'équation (28.15). Par exemple, on peut réécrire l'équation (28.9) de la manière suivante :

$$\begin{aligned} r &= ae + bg \\ &= (a \quad b \quad c \quad d) \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\ &= \begin{matrix} & e & f & g & h \\ a & + & \cdot & \cdot & \cdot \\ b & \cdot & \cdot & + & \cdot \\ c & \cdot & \cdot & \cdot & \cdot \\ d & \cdot & \cdot & \cdot & \cdot \end{matrix} \end{aligned}$$

La dernière expression utilise une notation abrégée dans laquelle « + » représente +1, « · » représente 0 et « - » représente -1. (A partir de maintenant, nous omettrons les étiquettes des lignes et des colonnes.) Avec cette notation, nous avons les équations suivantes pour les autres sous-matrices de la matrice résultat C :

$$\begin{aligned} s &= af + bh \\ &= \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \end{aligned}$$

$$\begin{aligned}
 t &= ce + dg \\
 &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix},
 \end{aligned}$$

$$\begin{aligned}
 u &= cf + dh \\
 &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
 \end{aligned}$$

On commence notre recherche d'un algorithme de multiplication de matrices plus rapide en observant que la sous-matrice s peut être calculée par $s = P_1 + P_2$, où P_1 et P_2 sont calculés chacun grâce à une multiplication de matrices :

$$\begin{aligned}
 P_1 &= A_1 B_1 \\
 &= a \cdot (f - h) \\
 &= af - ah \\
 &= \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},
 \end{aligned}$$

$$\begin{aligned}
 P_2 &= A_2 B_2 \\
 &= (a + b) \cdot h \\
 &= ah + bh \\
 &= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
 \end{aligned}$$

La matrice t peut être calculée de la même manière par $t = P_3 + P_4$, où

$$\begin{aligned}
 P_3 &= A_3 B_3 \\
 &= (c + d) \cdot e \\
 &= ce + de \\
 &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}
 \end{aligned}$$

et

$$\begin{aligned}
 P_4 &= A_4 B_4 \\
 &= d \cdot (g - e) \\
 &= dg - de \\
 &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}.
 \end{aligned}$$

Définissons un **terme essentiel** comme étant l'un des huit termes apparaissant dans le membre droit de l'une des équations (28.9)–(28.12). Nous avons jusqu'ici utilisé 4 produits pour calculer les deux sous-matrices s et t dont les termes essentiels sont af , bh , ce et dg . Notez que P_1 calcule le terme essentiel af , P_2 le terme essentiel bh , P_3 calcule ce et P_4 calcule dg . Donc, il nous reste à calculer les deux dernières sous-matrices r et u dont les termes essentiels sont ae , bg , cf et dh , en n'utilisant pas plus de trois produits supplémentaires. On essaye à présent d'innover avec P_5 , en calculant deux termes essentiels en même temps :

$$\begin{aligned}
 P_5 &= A_5 B_5 \\
 &= (a + d) \cdot (e + h) \\
 &= ae + ah + de + dh \\
 &= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.
 \end{aligned}$$

En plus de calculer les deux termes essentiels ae et dh , P_5 calcule les termes non essentiels ah et de , qui doivent être annulés d'une façon ou d'une autre. On peut utiliser P_4 et P_2 pour les annuler, mais deux autres termes non essentiels apparaissent alors :

$$\begin{aligned}
 P_5 + P_4 - P_2 &= ae + dh + dg - bh \\
 &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix}.
 \end{aligned}$$

Cependant, en ajoutant un produit supplémentaire

$$\begin{aligned}
 P_6 &= A_6 B_6 \\
 &= (b - d) \cdot (g + h) \\
 &= bg + bh - dg - dh \\
 &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix},
 \end{aligned}$$

on obtient

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= ae + bg \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Il est possible d'obtenir u de façon similaire à partir de P_5 en utilisant P_1 et P_3 pour déplacer les termes non essentiels de P_5 dans une direction différente :

$$\begin{aligned} P_5 + P_1 - P_3 &= ae + af - ce + dh \\ &= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

En soustrayant un produit supplémentaire

$$\begin{aligned} P_7 &= A_7 B_7 \\ &= (a - c) \cdot (e + f) \\ &= ae + af - ce - cf \\ &= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \end{aligned}$$

on obtient maintenant

$$\begin{aligned} u &= P_5 + P_1 - P_3 - P_7 \\ &= cf + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Les 7 produits de sous-matrices P_1, P_2, \dots, P_7 peuvent donc être utilisés pour calculer le produit $C = AB$, ce qui complète la description de la méthode de Strassen.

c) Discussion

D'un point de vue pratique, l'algorithme de Strassen est souvent peu recommandé pour la multiplication matricielle, et ce pour quatre raisons :

- 1) La facteur constant caché dans le temps d'exécution de l'algorithme est plus grand que celui de la méthode naïve à temps $\Theta(n^3)$.

- 2) Quand les matrices sont peu denses, on peut leur appliquer des méthodes spécifiques plus performantes.
- 3) L'algorithme de Strassen n'est pas aussi stable numériquement que la méthode basique.
- 4) Les sous-matrices créées aux niveaux de récursivité prennent de la place.

Les deux dernières raisons ont été mitigées vers 1990. Higham [145] a montré que l'on a surestimé la différence de stabilité numérique ; l'algorithme de Strassen est trop instable numériquement pour certaines applications, mais il est dans des limites acceptables pour d'autres. Bailey et al. [30] présentent des techniques pour diminuer les exigences en mémoire de l'algorithme de Strassen.

En pratique, les implémentations rapides de la multiplication matricielle pour les matrices denses emploient l'algorithme de Strassen pour des tailles de matrice supérieures à un « point d'équilibre » et repassent à la méthode basique dès que la taille de sous-problème passe en deçà du point d'équilibre. La valeur exacte du point d'équilibre dépend fortement du système. Des analyses, qui comptabilisent les opérations mais ignorent les effets des caches et de la pipelinisation, ont calculé des points d'équilibre pouvant descendre jusqu'à $n = 8$ (Higham [145]) ou $n = 12$ (Huss-Lederman et al. [163]). Les mesures empiriques donnent généralement des points d'équilibre plus élevés, certains descendant jusqu'à $n = 20$ ou à peu près. Pour un système donné, il est en principe facile de déterminer le point d'équilibre en faisant des essais.

En utilisant des techniques avancées qui sortent du champ de ce livre, on peut en fait multiplier des matrices $n \times n$ plus rapidement qu'en temps $\Theta(n^{\lg 7})$. Le meilleur majorant connu à ce jour vaut environ $O(n^{2,376})$. Le meilleur minorant connu est tout simplement la borne évidente $\Omega(n^2)$ (évidente car il faut remplir n^2 éléments de la matrice produit). Donc, on ne connaît toujours pas le niveau de difficulté exact de la multiplication matricielle.

Exercices

28.2.1 Utiliser l'algorithme de Strassen pour calculer le produit

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

Détailler les étapes du calcul.

28.2.2 Comment pourrait-on modifier l'algorithme de Strassen pour multiplier des matrices $n \times n$ pour lesquelles n n'est pas une puissance exacte de 2 ? Montrer que l'algorithme ainsi modifié s'exécute en temps $\Theta(n^{\lg 7})$.

28.2.3 Quel est le plus grand k tel que, s'il est possible de multiplier des matrices 3×3 à l'aide de k multiplications (sans supposer que la multiplication est commutative), alors il est possible de multiplier des matrices $n \times n$ en $o(n^{\lg 7})$? Quel serait le temps d'exécution de cet algorithme ?

28.2.4 V. Pan a découvert un moyen de multiplier des matrices 68×68 à l'aide de 132 464 multiplications, un moyen de multiplier des matrices 70×70 à l'aide de 143 640 multiplications, et un moyen de multiplier des matrices 72×72 à l'aide de 155 424 multiplications. Laquelle de ces méthodes aboutit au meilleur temps d'exécution asymptotique, quand on l'utilise dans un algorithme de multiplication matricielle diviser-pour-régner ? Comparer ce temps d'exécution avec celui de l'algorithme de Strassen.

28.2.5 A quelle vitesse peut-on multiplier une matrice $kn \times n$ par une matrice $n \times kn$, en se servant de l'algorithme de Strassen comme sous-programme ? Répondre à la même question en inversant l'ordre des matrices d'entrée.

28.2.6 Montrer comment multiplier les deux complexes $a + bi$ et $c + di$ en n'utilisant que trois multiplications de réels. L'algorithme devra prendre a, b, c et d en entrée et produire la partie réelle $ac - bd$ et la partie imaginaire $ad + bc$ séparément.

28.3 RÉOLUTION DE SYSTÈMES D'ÉQUATIONS LINÉAIRES

La résolution d'un ensemble d'équations linéaires simultanées est un problème fondamental, qui intervient dans de nombreuses applications. Un système linéaire peut s'exprimer sous la forme d'une équation de matrice, dans laquelle chaque élément de la matrice ou du vecteur appartient au même corps, en général celui des nombres réels \mathbf{R} . Cette section étudie la façon de résoudre un système d'équations linéaires par une méthode appelée décomposition LUP.

On commence avec un ensemble d'équations linéaires à n inconnues x_1, x_2, \dots, x_n :

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned} \tag{28.16}$$

Un ensemble de valeurs pour x_1, x_2, \dots, x_n qui vérifient toutes les équations (28.16) simultanément est appelé **solution** de ces équations. Dans cette section, on ne traitera que le cas où il existe exactement n équations pour n inconnues.

On peut facilement réécrire les équations (28.16) sous la forme de l'équation matrice-vecteur

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

ou encore, en posant $A = (a_{ij})$, $x = (x_j)$ et $b = (b_i)$,

$$Ax = b. \tag{28.17}$$

Si A est non singulière, elle possède un inverse A^{-1} , et

$$x = A^{-1}b \quad (28.18)$$

est le vecteur solution. On peut démontrer que le vecteur x est la solution unique de l'équation (28.17), de la façon suivante. S'il existait deux solutions x et x' , alors on aurait $Ax = Ax' = b$ et

$$\begin{aligned} x &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x'. \end{aligned}$$

Dans cette section, on s'intéressera surtout au cas où A est non singulière ou, ce qui revient au même (d'après le théorème 28.1), au cas où le rang de A est égal au nombre n d'inconnues. Il existe toutefois d'autres possibilités, qui méritent d'être brièvement discutées. Si le nombre d'équations est inférieur au nombre n d'inconnues, ou plus généralement si le rang de A est inférieur à n alors le système est *sous-déterminé*. Un système sous-déterminé possède en général un nombre infini de solutions, bien qu'il puisse aussi n'avoir aucune solution si les équations sont incohérentes. Si le nombre d'équations excède le nombre n d'inconnues, le système est *surdéterminé*, et il peut n'y avoir aucune solution. La recherche de bonnes solutions approchées pour un système d'équations linéaires surdéterminé est un problème important, étudié à la section 28.5.

Revenons à notre problème de résolution du système $Ax = b$ de n équations à n inconnues. Une approche pourrait consister à calculer A^{-1} , puis multiplier les deux membres de l'égalité par A^{-1} , pour obtenir $A^{-1}Ax = A^{-1}b$, soit $x = A^{-1}b$. En pratique, cette méthode a l'inconvénient d'être *instable numériquement*. Il existe heureusement une autre approche, la décomposition LUP, qui est numériquement stable et qui possède l'avantage supplémentaire d'être beaucoup plus rapide en pratique.

a) Vue d'ensemble de la décomposition LUP

Le principe de la décomposition LUP est de trouver trois matrices $n \times n$ L , U et P telles que

$$PA = LU, \quad (28.19)$$

où

- L est une matrice unitaire triangulaire inférieure,
- U est une matrice unitaire triangulaire supérieure, et
- P est une matrice de permutation.

Des matrices L , U et P qui vérifient l'équation (28.19) constituent une *décomposition LUP* de la matrice A . Nous allons montrer que toute matrice A non singulière possède une telle décomposition.

L'intérêt du calcul d'une décomposition LUP pour la matrice A est que les systèmes linéaires peuvent être résolus plus facilement quand ils sont triangulaires, ce qui est le cas des matrices L et U . Une fois trouvée une décomposition LUP pour A , on peut résoudre l'équation (28.17) $Ax = b$ en ne résolvant que des systèmes linéaires triangulaires, et ce de la façon suivante. Si l'on multiplie les deux membres de l'égalité $Ax = b$ par P , on obtient l'équation équivalente $PAx = Pb$ qui, d'après l'exercice 28.1.5, équivaut à permuter les équations (28.16). À l'aide de notre décomposition (28.19), on obtient

$$LUx = Pb .$$

Cette équation peut être résolue en résolvant deux systèmes linéaires triangulaires. Soit $y = Ux$, où x est le vecteur solution désiré. On commence par résoudre le système triangulaire inférieur

$$Ly = Pb \quad (28.20)$$

pour le vecteur inconnu y par la méthode dite de « substitution avant ». Une fois y trouvé, on résout le système triangulaire supérieur

$$Ux = y \quad (28.21)$$

pour l'inconnue x par la méthode de « substitution arrière ». Le vecteur x est notre solution de $Ax = b$, puisque la matrice de permutation P est inversible (exercice 28.1.5) :

$$\begin{aligned} Ax &= P^{-1}LUx \\ &= P^{-1}Ly \\ &= P^{-1}Pb \\ &= b . \end{aligned}$$

Notre prochaine étape consistera à montrer le fonctionnement des substitutions avant et arrière. Nous nous attaquerons ensuite au problème du calcul de la décomposition LUP elle-même.

b) Substitutions avant et arrière

La **substitution avant** permet de résoudre le système triangulaire inférieur (28.20) en temps $\Theta(n^2)$, à partir de L , P et b . Par commodité, on représente la permutation P de façon compacte, par un tableau $\pi[1..n]$. Pour $i = 1, 2, \dots, n$, l'élément $\pi[i]$ indique que $P_{i,\pi[i]} = 1$ et $P_{ij} = 0$ pour $j \neq \pi[i]$. Donc, PA contient $a_{\pi[i],j}$ à la ligne i et à la colonne j , et Pb contient $b_{\pi[i]}$ comme i ème élément. Puisque L est unitaire triangulaire inférieure, l'équation (28.20) peut être réécrite ainsi :

$$\begin{aligned} y_1 &= b_{\pi[1]} , \\ l_{21}y_1 + y_2 &= b_{\pi[2]} , \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]} , \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \dots + y_n &= b_{\pi[n]} . \end{aligned}$$

On peut résoudre y_1 directement, puisque la première équation nous dit que $y_1 = b_{\pi[1]}$. Une fois connue la valeur de y_1 , on peut la substituer dans la deuxième équation, ce qui donne

$$y_2 = b_{\pi[2]} - l_{21}y_1 .$$

A présent, on peut substituer y_1 et y_2 dans la troisième équation, pour obtenir

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) .$$

D'une manière générale, on substitue y_1, y_2, \dots, y_{i-1} « vers l'avant » dans la i ème équation pour trouver la valeur de y_i :

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j .$$

La **substitution arrière** ressemble à la substitution avant. Partant de U et y , on commence par résoudre la n ème équation puis on remonte jusqu'à la première. Comme pour la substitution avant, ce processus s'exécute en temps $\Theta(n^2)$. Puisque U est triangulaire supérieure, on peut réécrire le système (28.21) sous la forme :

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 , \\ u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 , \\ &\vdots \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} , \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} , \\ u_{n,n}x_n &= y_n . \end{aligned}$$

On peut donc connaître successivement les valeurs de x_n, x_{n-1}, \dots, x_1 de la façon suivante :

$$\begin{aligned} x_n &= y_n / u_{n,n} , \\ x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n) / u_{n-1,n-1} , \\ x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n)) / u_{n-2,n-2} , \\ &\vdots \end{aligned}$$

ou, plus généralement,

$$x_i = \left(y_i - \sum_{j=i+1}^n u_{ij}x_j \right) / u_{ii} .$$

Étant donnés P, L, U et b , la procédure RÉSOLUTION-LUP trouve la valeur de x en combinant substitution avant et substitution arrière. Le pseudo code suivant suppose que la dimension n se trouve dans l'attribut `lignes[L]` et que la matrice de permutation P est représentée par le tableau π .

RÉSOLUTION-LUP(L, U, π, b)

```

1   $n \leftarrow \text{lignes}[L]$ 
2  pour  $i \leftarrow 1$  à  $n$ 
3      faire  $y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j$ 
4  pour  $i \leftarrow n$  jusqu'à 1
5      faire  $x_i \leftarrow (y_i - \sum_{j=i+1}^n u_{ij}x_j) / u_{ii}$ 
6  retourner  $x$ 

```

La procédure RÉSOLUTION-LUP trouve la valeur de y *via* substitution avant aux lignes 2–3, puis elle trouve la valeur de x *via* substitution arrière aux lignes 4–5. Comme il existe une boucle implicite dans les sommations contenues dans chacune des boucles **pour**, le temps d'exécution est $\Theta(n^2)$.

Comme exemple d'application de cette méthode, considérons le système d'équations linéaires

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

où

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$

$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

que l'on souhaite résoudre pour trouver l'inconnue x . La décomposition LUP est

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix},$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(Le lecteur pourra vérifier que $PA = LU$.) *Via* substitution avant, on résout $Ly = Pb$ pour trouver y :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

ce qui donne

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

si l'on commence par calculer y_1 , puis y_2 , puis enfin y_3 . *Via* substitution arrière, on résout $Ux = y$ pour trouver x :

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix},$$

ce qui donne la réponse désirée

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

en calculant x_3 , puis x_2 , puis enfin x_1 .

c) Calcul d'une décomposition LU

Nous avons vu que, s'il est possible de calculer une décomposition LUP pour une matrice non singulière A , on peut utiliser des substitutions avant et arrière pour résoudre le système $Ax = b$ d'équations linéaires. Il reste à montrer comment on peut trouver efficacement une décomposition LUP pour A . Nous commençons par le cas où A est une matrice non singulière $n \times n$ et P est absent (ou, de façon équivalente, $P = I_n$). Dans ce cas, on doit trouver une factorisation $A = LU$. Les deux matrices L et U forment une **décomposition LU** de A .

Le processus par lequel on obtient une décomposition LU est appelé **élimination de Gauss**. On commence par soustraire des multiples de la première équation aux autres équations, de façon que la première variable soit éliminée de ces équations. Ensuite, on soustrait des multiples de la deuxième équation à la troisième équation et aux suivantes, pour qu'elles ne contiennent plus d'occurrences de la première et deuxième variable. On continue de la même manière, jusqu'à ce que le système ait une forme triangulaire supérieure; en fait, il s'agit de la matrice U . La matrice L est construite avec les multiplicateurs de lignes ayant permis d'éliminer les variables.

L'algorithme qui implémente cette stratégie est récursif. On souhaite construire une décomposition LU pour une matrice $n \times n$ non singulière A . Si $n = 1$, c'est terminé, puisqu'on peut choisir $L = I_1$ et $U = A$. Pour $n > 1$, on divise A en quatre

parties :

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right) \\ = \left(\begin{array}{c|c} a_{11} & {}^T w \\ \hline v & A' \end{array} \right),$$

où v est un vecteur colonne de taille $(n - 1)$, ${}^T w$ est un vecteur ligne de taille $(n - 1)$ et A' est une matrice $(n - 1) \times (n - 1)$. Ensuite, en faisant appel à l'algèbre des matrices (vérifiez les équations en effectuant la multiplication en sens inverse), on peut factoriser A de la manière suivante :

$$A = \left(\begin{array}{c|c} a_{11} & {}^T w \\ \hline v & A' \end{array} \right) \\ = \left(\begin{array}{c|c} 1 & 0 \\ \hline v/a_{11} & I_{n-1} \end{array} \right) \left(\begin{array}{c|c} a_{11} & {}^T w \\ \hline 0 & A' - v^T w/a_{11} \end{array} \right). \quad (28.22)$$

Les 0 de la première et deuxième matrices de la factorisation sont respectivement des vecteurs ligne et colonne de taille $n - 1$. Le terme $v^T w/a_{11}$, formé en prenant le produit extérieur de v et w puis en divisant chaque élément du résultat par a_{11} , est une matrice $(n - 1) \times (n - 1)$, compatible en taille avec la matrice A' dont elle est soustraite. La matrice $(n - 1) \times (n - 1)$ résultante

$$A' - v^T w/a_{11} \quad (28.23)$$

est appelée **complément de Schur** de A par rapport à a_{11} .

Nous affirmons que, si A est non singulière, alors le complément de Schur est non singulier lui aussi. Pourquoi ? Supposons que le complément de Schur, qui est une matrice $(n - 1) \times (n - 1)$, soit singulier. Alors, en vertu du théorème 28.1, il a un rang colonne strictement inférieur à $n - 1$. Comme les $n - 1$ éléments du bas de la première colonne de la matrice

$$\left(\begin{array}{c|c} a_{11} & w^T \\ \hline 0 & A' - v w^T/a_{11} \end{array} \right)$$

sont tous des 0, les $n - 1$ lignes du bas de cette matrice ont forcément un rang ligne qui est strictement inférieur à $n - 1$. Le rang ligne de la matrice complète est donc strictement inférieur à n . En appliquant l'exercice 28.1.10 à l'équation (28.22), A a un rang strictement inférieur à n ; en vertu du théorème 28.1, on arrive alors à la contradiction que A est singulière.

Comme le complément de Schur est non singulier, on peut maintenant calculer récursivement sa décomposition LU. Posons

$$A' - v^T w/a_{11} = L' U',$$

où L' est triangulaire inférieure unitaire, et U' est triangulaire supérieure. Alors, en faisant appel aux propriétés de l'algèbre des matrices, on a

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & {}^T w \\ 0 & A' - v {}^T w / a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & {}^T w \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & {}^T w \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

ce qui nous donne notre décomposition LU. (Notez que, comme L' est triangulaire inférieure unitaire, L l'est également, et comme U' est triangulaire supérieure, U l'est aussi.)

Bien sûr, si $a_{11} = 0$, cette méthode ne fonctionne pas, puisqu'elle effectue une division par 0. Elle ne marche pas non plus si l'élément supérieur le plus à gauche du complément de Schur $A' - v {}^T w / a_{11}$ a la valeur 0, puisqu'il sert de diviseur à la prochaine étape de la récursivité. Les éléments qui jouent le rôle de diviseurs pendant la décomposition LU sont appelés **pivots**, et ils occupent la diagonale de la matrice U . La raison de la présence d'une matrice de permutation P dans la décomposition LUP est qu'elle évite de diviser par des éléments nuls. L'utilisation de permutations pour éviter la division par 0 (ou par de petits nombres) est appelé **pivotement**.

Une classe de matrices importante pour laquelle la décomposition LU fonctionne toujours correctement, est la classe des matrices symétriques définies positives. Ces matrices n'ont pas besoin de pivotement, et la stratégie récursive que nous venons de résumer peut s'utiliser sans que l'on craigne de diviser par 0. Nous prouverons ce résultat, avec plusieurs autres, à la section 28.5.

Notre code de décomposition LU d'une matrice A suit la stratégie récursive, sauf que la récursivité a été remplacée par une boucle itérative. (Cette transformation est une optimisation classique pour une procédure « récursive terminale », c'est-à-dire une procédure dont la dernière opération est un appel récursif à elle-même.) Le code suppose que la dimension de A est mémorisée dans l'attribut *lignes*[A]. Comme nous savons que la matrice de sortie U possède des 0 sous la diagonale, et comme RÉSOLUTION-LU ne regarde pas ces éléments, le code n'a pas besoin de les remplir. De même, comme la matrice de sortie L comporte des 1 sur sa diagonale et des 0 au-dessus, ces éléments ne sont pas remplis. Le code ne calcule donc que les éléments « significatifs » de L et U .

DÉCOMPOSITION-LU(A)

```

1   $n \leftarrow \text{lignes}[A]$ 
2  pour  $k \leftarrow 1$  à  $n$ 
3      faire  $u_{kk} \leftarrow a_{kk}$ 
4          pour  $i \leftarrow k + 1$  à  $n$ 
5              faire  $l_{ik} \leftarrow a_{ik}/u_{kk}$   $\triangleright l_{ik}$  contient  $v_i$ 
6                   $u_{ki} \leftarrow a_{ki}$   $\triangleright u_{ki}$  contient  ${}^T w_i$ 
7          pour  $i \leftarrow k + 1$  à  $n$ 
8              faire pour  $j \leftarrow k + 1$  à  $n$ 
9                  faire  $a_{ij} \leftarrow a_{ij} - l_{ik}u_{kj}$ 
10         retourner  $L$  et  $U$ 

```

La boucle **pour** extérieure qui commence à la ligne 2 est itérée une fois pour chaque étape récursive. À l'intérieur de cette boucle, le pivot est calculé comme étant $u_{kk} = a_{kk}$ à la ligne 3. À l'intérieur de la boucle **pour** des lignes 4–6 (qui ne s'exécute pas quand $k = n$), les vecteurs v et ${}^T w$ sont utilisés pour mettre à jour L et U . Les éléments du vecteur v sont déterminés à la ligne 5, où v_i est stocké dans l_{ik} , et les éléments du vecteur ${}^T w$ sont déterminés à la ligne 6, où ${}^T w_i$ est stocké dans u_{ki} . Enfin, les éléments du complément de Schur sont calculés aux lignes 7–9 puis remis dans la matrice A . (Nous n'avons pas besoin de diviser par a_{kk} en ligne 9, car cela a déjà été fait quand nous avons calculé l_{ik} en ligne 5.) Comme la ligne 9 est au troisième niveau de boucle, DÉCOMPOSITION-LU s'exécute en temps $\Theta(n^3)$.

La figure 28.1 illustre l'action de DÉCOMPOSITION-LU. Elle montre une optimisation classique de la procédure, dans laquelle les éléments significatifs de L et U sont stockés « sur place » dans la matrice A . Autrement dit, on peut créer une correspondance entre chaque élément a_{ij} et l_{ij} (si $i > j$) ou u_{ij} (si $i \leq j$), puis mettre à jour la matrice A de sorte qu'elle contienne L et U à la fin de la procédure. Le pseudo code pour cette optimisation est obtenu à partir du pseudo code précédent, en remplaçant simplement chaque référence à l ou u par a ; il est facile de vérifier que cette transformation préserve la validité de l'algorithme.

d) Calcul d'une décomposition LUP

D'une façon générale, quand on résout un système d'équations linéaires $Ax = b$, on doit pivoter sur des éléments non diagonaux de A pour éviter les divisions par 0. La division par 0 n'est pas la seule à être indésirable ; la division par une petite valeur l'est également, même si A n'est pas singulière, à cause de l'instabilité numérique qui peut apparaître lors du calcul. On essaye donc de pivoter sur une grande valeur.

Le traitement mathématique de la décomposition LUP est similaire à celui de la décomposition LU. Rappelons-nous que l'on a une matrice $n \times n$ non singulière A et que l'on veut trouver une matrice de permutation P , une matrice triangulaire inférieure L et une matrice triangulaire supérieure U telles que $PA = LU$. Avant de partitionner la matrice A , comme nous l'avons fait pour la décomposition LU, on

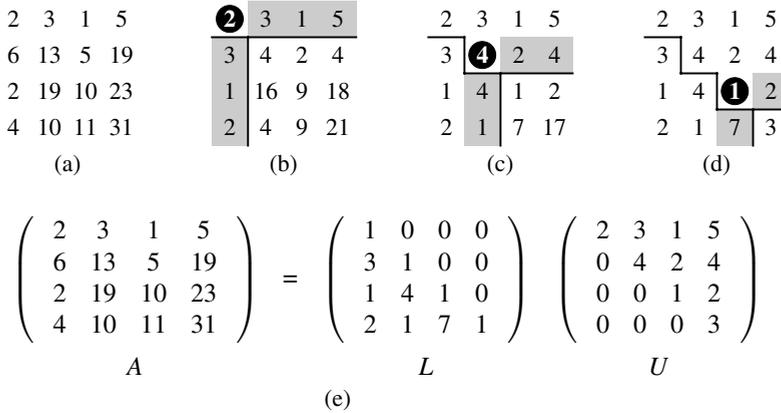


Figure 28.1 L'action de DÉCOMPOSITION-LU. (a) La matrice A. (b) L'élément $a_{11} = 2$ représenté en noir est le pivot, la colonne en gris est v/a_{11} , et la ligne en gris est ${}^T w$. Les éléments de U calculés jusqu'ici se trouvent au-dessus de la ligne horizontale, et ceux de L sont à gauche de la ligne verticale. Le complément de Schur, la matrice $A' - v{}^T w/a_{11}$, occupe le partie inférieure droite. (c) On agit ensuite sur le complément de Schur produit à la partie (b). L'élément $a_{22} = 4$ en noir est le pivot, et la colonne et la ligne en gris représentent respectivement v/a_{22} et ${}^T w$ (dans le partitionnement du complément de Schur). Les lignes divisent la matrice en les éléments de U calculés jusqu'ici (en haut), les éléments de L calculés jusqu'ici (à gauche) et le nouveau complément de Schur (en bas à droite). (d) L'étape suivante termine la factorisation. (L'élément 3 du nouveau complément de Schur finira par appartenir à U en fin de récursivité.) (e) La factorisation $A = LU$.

déplace un élément non nul, disons a_{k1} , de la première colonne vers la position (1, 1) de la matrice. (Si la première colonne ne contient que des 0, alors A est singulière, car son déterminant a pour valeur 0, d'après les théorèmes 28.4 et 28.5.) Pour préserver l'ensemble d'équations, on échange la ligne 1 avec la ligne k, ce qui équivaut à multiplier à gauche la matrice A par une matrice de permutation Q (exercice 28.1.5). On peut donc écrire QA ainsi :

$$QA = \begin{pmatrix} a_{k1} & {}^T w \\ v & A' \end{pmatrix},$$

où $v = {}^T (a_{21}, a_{31}, \dots, a_{n1})$, sauf que a_{11} remplace a_{k1} ; ${}^T w = (a_{k2}, a_{k3}, \dots, a_{kn})$; et A' est une matrice $(n - 1) \times (n - 1)$. Puisque $a_{k1} \neq 0$, on peut à présent appliquer les mêmes opérations d'algèbre linéaire que pour la décomposition LU, mais en étant sûr de ne jamais rencontrer de division par 0 :

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & {}^T w \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & {}^T w \\ 0 & A' - v{}^T w/a_{k1} \end{pmatrix}. \end{aligned}$$

Comme nous l'avons vu pour la décomposition LU, si A est non singulière, alors le complément de Schur $A' - vw^T/a_{k1}$ est non singulier lui aussi. Donc, l'on peut par récurrence lui trouver une décomposition LUP constituée d'une matrice unitaire triangulaire inférieure L' , d'une matrice triangulaire supérieure U' et d'une matrice de permutation P' telles que

$$P'(A' - vw^T/a_{k1}) = L'U'.$$

On définit

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q,$$

qui est une matrice de permutation, puisqu'elle est le produit de deux matrices de permutation (exercice 28.1.5). On a maintenant

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & P'(A' - vw^T/a_{k1}) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

ce qui aboutit à la décomposition LUP. L' étant une matrice unitaire triangulaire inférieure, L l'est aussi ; de même, U' étant triangulaire supérieure, U l'est également.

Notez que dans ce calcul, contrairement à celui de la décomposition LU, le vecteur colonne v/a_{k1} et le complément de Schur $A' - v^T w/a_{k1}$ doivent tous les deux être multipliés par la matrice de permutation P' .

Comme pour DÉCOMPOSITION-LU, notre pseudo code implémentant la décomposition LUP remplace la récursivité par une boucle itérative. Comme amélioration par rapport à l'implémentation directe de la récursivité, on gère dynamiquement la matrice de permutation P en tant que tableau π , où $\pi[i] = j$ signifie que la i ème ligne de P contient un 1 dans la colonne j . On s'arrange également, au niveau du code, pour calculer L et U « sur place » dans la matrice A . Donc, quand la procédure se termine,

$$a_{ij} = \begin{cases} l_{ij} & \text{si } i > j, \\ u_{ij} & \text{si } i \leq j. \end{cases}$$

DÉCOMPOSITION-LUP(A)

```

1  n ← lignes[A]
2  pour i ← 1 à n
3      faire π[i] ← i
4  pour k ← 1 à n
5      faire p ← 0
6          pour i ← k à n
7              faire si |aik| > p
8                  alors p ← |aik|
9                      k' ← i
10         si p = 0
11             alors erreur « matrice singulière »
12         permuter π[k] ↔ π[k']
13         pour i ← 1 à n
14             faire permuter aki ↔ ak'i
15         pour i ← k + 1 à n
16             faire aik ← aik/akk
17             pour j ← k + 1 à n
18                 faire aij ← aij - aikakj

```

La figure 28.2 illustre comment une matrice est factorisée par DÉCOMPOSITION-LUP. Le tableau π est initialisé aux lignes 2–3 pour représenter la permutation identité. La boucle **pour** extérieure commençant à la ligne 4 implémente la récursivité. A chaque passage dans la boucle extérieure, les lignes 5–9 déterminent l'élément $a_{k'k}$ de plus grande valeur absolue parmi ceux de la première colonne (colonne k) de la matrice $(n - k + 1) \times (n - k + 1)$ dont on doit trouver une décomposition LU. Si tous les éléments de la première colonne courante valent zéro, les lignes 10–11 signalent que la matrice est singulière. Pour pivoter, on échange $\pi[k']$ avec $\pi[k]$ à la ligne 12 et on échange les k ième et k' ième lignes de A en 13–14, ce qui fait de a_{kk} l'élément pivot. (On échange les lignes tout entières car, dans la méthode de calcul précédente, non seulement $A' - v^T w/a_{k1}$ est multipliée par P' , mais v/a_{k1} l'est aussi.) Enfin, le complément de Schur est calculé par les lignes 15–18 d'une façon pratiquement identique à celle des lignes 4–9 de DÉCOMPOSITION-LU, sauf qu'ici l'opération est écrite pour fonctionner « sur place ».

À cause de sa structure à trois boucles imbriquées, le temps d'exécution de DÉCOMPOSITION-LUP est $\Theta(n^3)$, comme pour DÉCOMPOSITION-LU. Donc, le pivotement ne coûte au plus qu'un facteur multiplicatif constant.

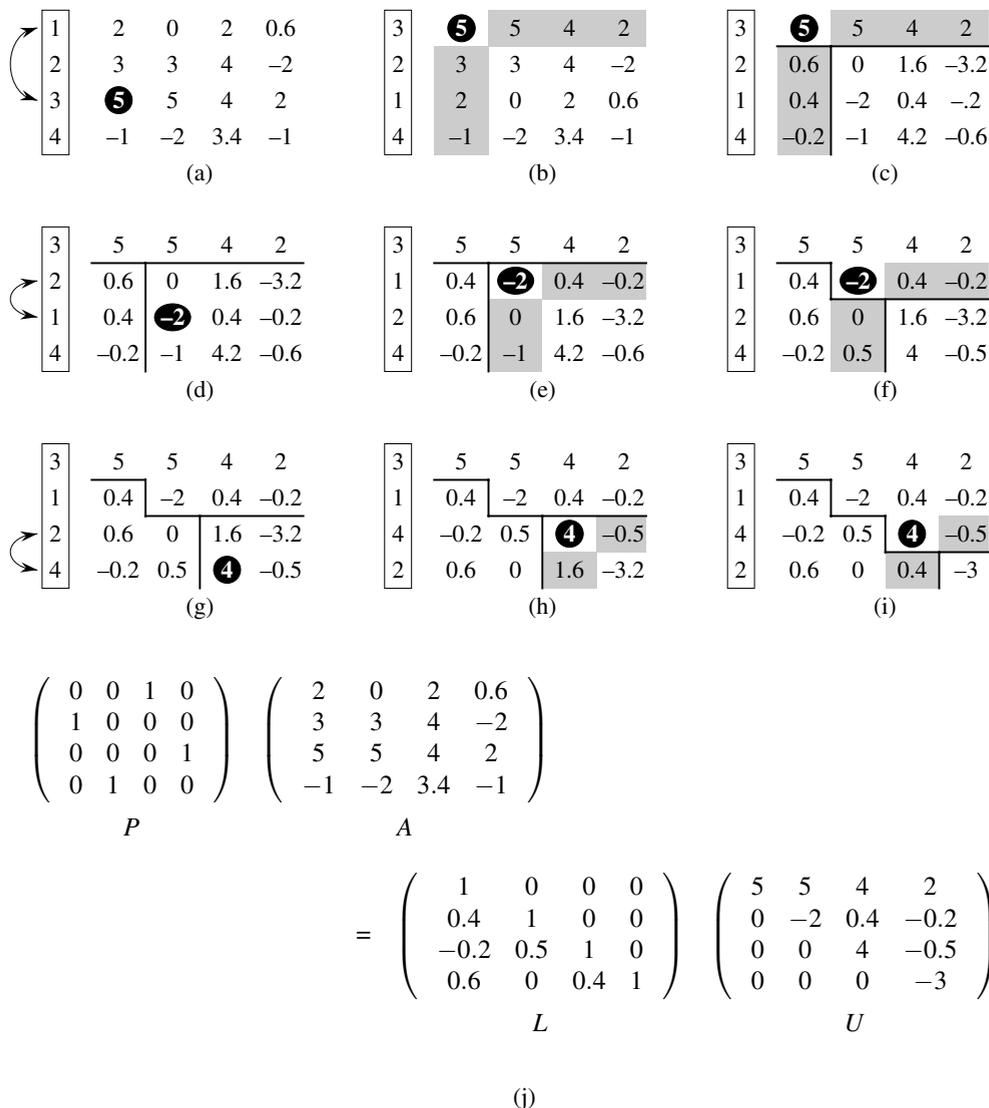


Figure 28.2 L'action de DÉCOMPOSITION-LUP. (a) La matrice d'entrée A avec la permutation identité des lignes à gauche. La première étape de l'algorithme choisit l'élément 5 en noir dans la troisième ligne comme pivot pour la première colonne. (b) Les lignes 1 et 3 sont échangées, et la permutation est mise à jour. La colonne et les lignes en gris représentent v et ${}^T w$. (c) Le vecteur v est remplacé par $v/5$, et la partie en bas à droite de la matrice est remplacée par le complément de Schur. Les traits divisent la matrice en trois régions : les éléments de U (en haut), ceux de L (à gauche) et ceux du complément de Schur (en bas à droite). (d)–(f) La deuxième étape. (g)–(i) La troisième étape. Aucune modification supplémentaire n'a lieu durant la quatrième et dernière étape. (j) La décomposition LUP $PA = LU$.

Exercices

28.3.1 Résoudre l'équation

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

via substitution avant.

28.3.2 Trouver une décomposition LU de la matrice

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

28.3.3 Résoudre l'équation

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

à l'aide d'une décomposition LUP.

28.3.4 Décrire la décomposition LUP d'une matrice diagonale.

28.3.5 Décrire la décomposition LUP d'une matrice de permutation A , et démontrer qu'elle est unique.

28.3.6 Montrer que, pour tout $n \geq 1$, il existe une matrice singulière $n \times n$ qui a une décomposition LU.

28.3.7 Dans DÉCOMPOSITION-LU, est-il nécessaire d'effectuer l'itération de la boucle **pour** la plus extérieure quand $k = n$? Et dans DÉCOMPOSITION-LUP?

28.4 INVERSION DES MATRICES

Bien qu'en pratique les matrices inverses ne soient pas utilisées pour résoudre les systèmes d'équations linéaires, car on préfère employer des techniques plus stables numériquement comme la décomposition LUP, il est parfois nécessaire de calculer l'inverse d'une matrice. Dans cette section, on verra comment utiliser la décomposition LUP pour calculer l'inverse d'une matrice. Nous montrerons aussi que la multiplication matricielle et le calcul de l'inverse d'une matrice sont des problèmes de difficulté équivalente, en ce sens que (moyennant certaines conditions techniques) on peut employer un algorithme conçu pour l'un pour résoudre l'autre avec le même temps d'exécution asymptotique. On peut donc utiliser l'algorithme de Strassen de la multiplication matricielle pour qu'il calcule l'inverse d'une matrice. En fait, l'article

originel de Strassen fut motivé par le problème consistant à montrer qu'un ensemble d'équations linéaires pouvait être résolu plus rapidement qu'avec la méthode habituelle.

a) *Calcul de l'inverse d'une matrice à partir d'une décomposition LUP*

Supposons qu'on dispose d'une décomposition LUP d'une matrice A sous la forme de trois matrices L , U et P telles que $PA = LU$. À l'aide de RÉSOLUTION-LU, on peut résoudre une équation de la forme $Ax = b$ en temps $\Theta(n^2)$. Comme la décomposition LUP dépend de A mais pas de b , on peut résoudre un deuxième ensemble d'équations de la forme $Ax = b'$ avec un temps supplémentaire de $\Theta(n^2)$. En général, une fois connue la décomposition LUP de A , il est possible de résoudre en temps $\Theta(kn^2)$, k versions de l'équation $Ax = b$ qui ne diffèrent que par b .

L'équation

$$AX = I_n \quad (28.24)$$

peut être considérée comme un ensemble de n équations distinctes de la forme $Ax = b$. Ces équations définissent la matrice X comme l'inverse de A . Plus précisément, notons X_i la i ème colonne de X , et rappelons-nous que le vecteur unitaire e_i est la i ème colonne de I_n . L'équation (28.24) peut alors être résolue pour X en utilisant la décomposition LUP de A pour résoudre chaque équation

$$AX_i = e_i$$

séparément pour l'inconnue X_i . Chacune des n colonnes X_i peut être trouvée en temps $\Theta(n^2)$, et le calcul de X à partir de la décomposition LUP de A prend donc un temps $\Theta(n^3)$. Comme la décomposition LUP de A peut se calculer en temps $\Theta(n^3)$, l'inverse A^{-1} d'une matrice A peut être déterminé en temps $\Theta(n^3)$.

b) *Multiplication matricielle et inversion de matrice*

Montrons à présent que les accélérations théoriques obtenues pour la multiplication des matrices peuvent se traduire en accélération pour l'inversion des matrices. En fait, nous démontrons quelque chose de plus fort : l'inversion et la multiplication des matrices sont deux opérations équivalentes, au sens suivant. Si $M(n)$ désigne le temps nécessaire à la multiplication de deux matrices $n \times n$, alors il y a moyen d'inverser une matrice $n \times n$ en temps $O(M(n))$. En outre, si $I(n)$ désigne le temps nécessaire à l'inversion d'une matrice $n \times n$ non singulière, alors il y a moyen de multiplier deux matrices $n \times n$ en temps $O(I(n))$. Ce résultat se démontre sous la forme de deux théorèmes distincts.

Théorème 28.7 (La multiplication n'est pas plus complexe que l'inversion) *Si l'on peut inverser une matrice $n \times n$ en temps $I(n)$, où $I(n) = \Omega(n^2)$ et $I(n)$ satisfait à la condition de régularité $I(3n) = O(I(n))$, alors il est possible de multiplier deux matrices $n \times n$ en temps $O(I(n))$.*

Démonstration : Soient A et B deux matrices $n \times n$ dont on souhaite calculer le produit C . On définit la matrice D de taille $3n \times 3n$ par

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

L'inverse de D est

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

et on peut donc calculer le produit AB en prenant la sous-matrice de D^{-1} située dans le coin supérieur droit et de taille $n \times n$.

On peut construire la matrice D en temps $\Theta(n^2) = O(I(n))$, et on peut inverser D en temps $O(I(3n)) = O(I(n))$, d'après la condition de régularité sur $I(n)$. On a donc

$$M(n) = O(I(n)).$$

□

Notez que $I(n)$ satisfait à la condition de régularité à partir dès que $I(n) = \Theta(n^c \lg^d n)$ pour deux constantes $c > 0$ et $d \geq 0$ quelconques. Pour montrer que l'inversion matricielle n'est pas plus complexe que la multiplication matricielle, on fait appel à certaines propriétés des matrices définies positives qui seront prouvées à la section 28.5.

Théorème 28.8 (L'inversion n'est pas plus complexe que la multiplication) *Supposons qu'on sache multiplier deux matrices $n \times n$ à valeurs réelles en temps $M(n)$, où $M(n) = \Omega(n^2)$ et $M(n)$ satisfait aux deux conditions de régularité $M(n+k) = O(M(n))$ pour un k quelconque de l'intervalle $0 \leq k \leq n$ et $M(n/2) \leq cM(n)$ pour une certaine constante $c < 1/2$. On peut alors calculer l'inverse d'une matrice réelle non singulière de taille $n \times n$ en temps $O(M(n))$.*

Démonstration : On peut supposer que n est une puissance exacte de 2, puisqu'on a

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

pour tout $k > 0$. Donc, en choisissant k pour que $n+k$ soit une puissance de 2, on agrandit la matrice à une taille égale à la puissance de 2 suivante, et on obtient la réponse souhaitée A^{-1} à partir de la réponse au problème agrandi. La première condition de régularité sur $M(n)$ garantit que cet agrandissement n'augmente pas le temps d'exécution de plus d'un facteur constant.

Pour l'instant, faisons l'hypothèse que la matrice A $n \times n$ est symétrique et définie positive. On partitionne A en quatre sous-matrices $n/2 \times n/2$:

$$A = \begin{pmatrix} B & {}^T C \\ C & D \end{pmatrix}. \quad (28.25)$$

Alors, si l'on note

$$S = D - CB^{-1}{}^T C \quad (28.26)$$

le complément de Schur de A par rapport à B (nous reviendrons sur cette forme de complément de Schur à la section 28.5), on obtient

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1T}CS^{-1}CB^{-1} & -B^{-1T}CS^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \quad (28.27)$$

puisque $AA^{-1} = I_n$, comme on peut le vérifier en effectuant la multiplication. Les matrices B^{-1} et S^{-1} existent si A est symétrique et définie positive, d'après les lemmes 28.9, 28.10 et 28.11 de la section 28.5, car B et S sont symétriques et définies positives. D'après l'exercice 28.1.2, $B^{-1T}C = {}^T(CB^{-1})$ et $B^{-1T}CS^{-1} = {}^T(S^{-1}CB^{-1})$. Les équations (28.26) et (28.27) peuvent donc servir à spécifier un algorithme récursif mettant en jeu 4 multiplications de matrices $n/2 \times n/2$:

$$\begin{aligned} & C \cdot B^{-1}, \\ & (CB^{-1}) \cdot {}^T C, \\ & S^{-1} \cdot (CB^{-1}), \\ & {}^T(CB^{-1}) \cdot (S^{-1}CB^{-1}). \end{aligned}$$

Donc, on peut inverser une matrice $n \times n$ symétrique définie positive en inversant deux matrices $n/2 \times n/2$ (B et S), puis en effectuant ces quatre multiplications de matrices $n/2 \times n/2$ (ce que l'on peut faire avec un algorithme pour matrices $n \times n$), plus un coût additionnel de $O(n^2)$ pour extraire des sous-matrices de A et faire un nombre constant d'additions et de soustractions sur ces matrices $n/2 \times n/2$. On a donc la récurrence

$$\begin{aligned} I(n) & \leq 2I(n/2) + 4M(n) + O(n^2) \\ & = 2I(n/2) + \Theta(M(n)) \\ & = O(M(n)). \end{aligned}$$

La deuxième ligne se justifie par le fait que $M(n) = \Omega(n^2)$; la troisième ligne découle du fait que la seconde condition de régularité donnée dans l'énoncé du théorème permet d'appliquer le cas 3 du théorème général (théorème 4.1).

Il reste montrer qu'on peut atteindre pour l'inversion des matrices le même temps d'exécution asymptotique que pour la multiplication des matrices, quand A est inversible sans être symétrique et définie positive. Le principe est que, pour une matrice A non singulière, la matrice TAA est symétrique (d'après l'exercice 28.1.2) et définie positive (d'après le théorème 28.6). L'astuce consiste alors à ramener le problème de l'inversion de A au problème de l'inversion de TAA .

Ce déplacement du problème se fonde sur l'observation que, si A est une matrice $n \times n$ non singulière, on a

$$A^{-1} = ({}^TAA)^{-1}{}^T A,$$

car $(({}^TAA)^{-1}{}^T A)A = ({}^TAA)^{-1}({}^TAA) = I_n$ et une matrice inverse est unique. On peut donc calculer A^{-1} en commençant par multiplier ${}^T A$ par A pour obtenir TAA , puis en inversant la matrice symétrique définie positive TAA à l'aide de l'algorithme diviser-pour-régner précédent, puis enfin en multipliant le résultat par ${}^T A$. Chacune de ces trois étapes nécessite un temps $O(M(n))$, et il est donc possible d'inverser n'importe quelle matrice non singulière à éléments réel en temps $O(M(n))$. \square

La démonstration du théorème 28.8 suggère une manière de résoudre l'équation $Ax = b$ via décomposition LU sans pivotement, du moment que A n'est pas singulière.

On multiplie les deux membres de l'équation par ${}^T A$, ce qui donne $({}^T A A)x = {}^T A b$. Cette transformation ne modifie pas la solution x , puisque ${}^T A$ est inversible et que l'on peut donc factoriser la matrice symétrique définie positive ${}^T A A$ en calculant une décomposition LU. On utilise ensuite les substitutions avant et arrière pour résoudre l'équation pour x avec un membre droit égal à ${}^T A b$. Bien que cette méthode soit théoriquement correcte, la procédure DÉCOMPOSITION-LUP fonctionne beaucoup mieux en pratique. La décomposition LUP demande moins d'opérations arithmétiques et offre des propriétés numériques plutôt meilleures.

Exercices

28.4.1 Soit $M(n)$ le temps requis pour la multiplication de matrices $n \times n$, et soit $S(n)$ le temps nécessaire pour élever au carré une matrice $n \times n$. Montrer que la multiplication et l'élevation au carré ont essentiellement la même complexité : un algorithme de multiplication matricielle à temps $M(n)$ implique un algorithme d'élevation au carré à temps $O(M(n))$, et un algorithme d'élevation au carré à temps $S(n)$ implique un algorithme de multiplication à temps $O(S(n))$.

28.4.2 Soit $M(n)$ le temps requis pour la multiplication de matrices $n \times n$, et soit $L(n)$ le temps nécessaire pour calculer la décomposition LUP d'une matrice $n \times n$. Montrer que la multiplication des matrices et le calcul d'une décomposition LUP ont essentiellement la même complexité : un algorithme de multiplication matricielle à temps $M(n)$ implique un algorithme de décomposition LUP à temps $O(M(n))$, et un algorithme de décomposition LUP à temps $L(n)$ implique un algorithme de multiplication à temps $O(L(n))$.

28.4.3 Soit $M(n)$ le temps requis pour la multiplication de matrices $n \times n$, et soit $D(n)$ le temps nécessaire pour trouver le déterminant d'une matrice $n \times n$. Montrer que la multiplication des matrices et le calcul d'un déterminant ont essentiellement la même complexité : un algorithme de multiplication matricielle à temps $M(n)$ implique un algorithme de déterminant à temps $O(M(n))$, et un algorithme de déterminant à temps $D(n)$ implique un algorithme de multiplication à temps $O(D(n))$.

28.4.4 Soit $M(n)$ le temps requis pour la multiplication de matrices booléennes $n \times n$, et soit $T(n)$ le temps nécessaire pour trouver la fermeture transitive de matrices booléennes $n \times n$. (Voir section 25.2.) Montrer qu'un algorithme de multiplication de matrices booléennes à temps $M(n)$ implique un algorithme de fermeture transitive à temps $O(M(n) \lg n)$, et qu'un algorithme de fermeture transitive à temps $T(n)$ implique un algorithme de multiplication de matrices booléennes à temps $O(T(n))$.

28.4.5 L'algorithme d'inversion de matrice basé sur le théorème 28.8 fonctionne-t-il quand les éléments de la matrice appartiennent au corps des entiers modulo 2 ? Pourquoi ?

28.4.6 * Généraliser l'algorithme d'inversion de matrice décrit au théorème 28.8 pour qu'il gère les matrices de nombres complexes, et montrer que votre généralisation fonctionne correctement. (*Conseil* : Au lieu de la transposée de A , utiliser la *transposée conjuguée* A^* ,

obtenue à partir de la transposée de A en remplaçant chaque élément par le conjugué complexe. Au lieu de matrices symétriques, considérer des matrices *hermitiennes*, qui sont des matrices A telles que $A = A^*$.)

28.5 MATRICES SYMÉTRIQUES DÉFINIES POSITIVES ET APPROXIMATION DES MOINDRES CARRÉS

Les matrices symétriques définies positives possèdent de nombreuses propriétés intéressantes. Par exemple, elles sont non singulières, et on peut leur appliquer une décomposition LU sans se soucier d'une éventuelle division par 0. Dans cette section, nous allons démontrer plusieurs autres propriétés importantes des matrices symétriques définies positives, et nous montrerons une application intéressante permettant de trouver une courbe approchée d'un ensemble de points par la méthode des moindres carrés.

La première propriété est peut-être la plus fondamentale.

Lemme 28.9 *Toute matrice définie positive est non singulière.*

Démonstration : On suppose qu'une matrice A donnée est singulière. D'après le corollaire 28.3, il existe un vecteur x non nul tel que $Ax = 0$. Donc, $x^T Ax = 0$ et A ne peut pas être définie positive. \square

Il est moins facile de prouver qu'une décomposition LU sur une matrice symétrique définie positive A peut s'effectuer sans division par 0. On commencera par démontrer des propriétés sur certaines sous-matrices de A . On définit la k ième *sous-matrice initiale* de A comme étant la matrice A_k composée de l'intersection des k premières lignes et des k premières colonnes de A .

Lemme 28.10 *Si A est une matrice symétrique définie positive, alors toute sous-matrice initiale de A est symétrique et définie positive.*

Démonstration : Le fait que toute sous-matrice initiale A_k soit symétrique est évident. Pour prouver que A_k est définie positive, nous allons supposer le contraire et aboutir à une contradiction. Si A_k n'est pas définie positive, alors il existe un vecteur de taille k $x_k \neq 0$ tel que $x_k^T A_k x_k \leq 0$. Si A est une matrice $n \times n$, on définit le vecteur de taille n $x = (x_k^T \ 0)^T$, où il y a $n - k$ 0 après x_k . On a alors

$$\begin{aligned} x^T Ax &= \begin{pmatrix} x_k^T & 0 \end{pmatrix} \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} x_k^T & 0 \end{pmatrix} \begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^T A_k x_k \\ &\leq 0, \end{aligned}$$

ce qui contredit que A est définie positive. \square

Passons à quelques propriétés fondamentales du complément de Schur. Soit A une matrice symétrique définie positive, et soit A_k une sous-matrice initiale $k \times k$ de A . On partitionne A ainsi :

$$A = \begin{pmatrix} A_k & {}^T B \\ B & C \end{pmatrix}. \quad (28.28)$$

En généralisant la définition (28.23), on définit le **complément de Schur** de A par rapport à A_k comme étant

$$S = C - BA_k^{-1}{}^T B. \quad (28.29)$$

(D'après le lemme 28.10, A_k est symétrique et définie positive ; donc A_k^{-1} existe d'après le lemme 28.9, et S est bien défini.) Notez que notre précédente définition (28.23) du complément de Schur est cohérente avec la définition (28.29), en prenant $k = 1$.

Le lemme suivant montre que les matrices complément de Schur de matrices symétriques définies positives sont elles-mêmes symétriques et définies positives. Ce résultat nous a servi pour le théorème 28.8, et son corollaire est nécessaire pour démontrer la validité de la décomposition LU pour les matrices symétriques définies positives.

Lemme 28.11 (Lemme du complément de Schur) *Si A est une matrice symétrique définie positive et si A_k est une sous-matrice initiale de A de taille $k \times k$, alors le complément de Schur de A par rapport à A_k est une matrice symétrique définie positive.*

Démonstration : A étant symétrique, il en est de même de la sous-matrice C . D'après l'exercice 28.1.8, le produit $BA_k^{-1}{}^T B$ est symétrique et, d'après l'exercice 28.1.1, S est symétrique.

Il reste à montrer que S est définie positive. On considère la partition de A donnée à l'équation (28.28). Pour un vecteur x non nul quelconque, on a ${}^T x A x > 0$ de par l'hypothèse que A est définie positive. Coupons x en deux sous-vecteurs y et z , respectivement compatibles avec A_k et C . Comme A_k^{-1} existe, on a

$$\begin{aligned} x^T A x &= \begin{pmatrix} y^T & z^T \end{pmatrix} \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= \begin{pmatrix} y^T & z^T \end{pmatrix} \begin{pmatrix} A_k y + B^T z \\ B y + C z \end{pmatrix} \\ &= y^T A_k y + y^T B^T z + z^T B y + z^T C z \\ &= (y + A_k^{-1} B^T z)^T A_k (y + A_k^{-1} B^T z) + z^T (C - BA_k^{-1} B^T) z, \end{aligned} \quad (28.30)$$

par la magie des matrices. (On peut vérifier en remultipliant les termes.) Cette dernière équation revient à « compléter le carré » de la forme quadratique. (Voir exercice 28.5.2.)

Comme ${}^T x A x > 0$ est vrai pour tout x non nul, on prend un z non nul quelconque puis on choisit $y = -A_k^{-1} B^T z$, ce qui annule le premier terme de l'équation (28.30), pour laisser

$${}^T z (C - BA_k^{-1} B^T) z = {}^T z S z$$

comme valeur de l'expression. Pour un $z \neq 0$ quelconque, on a donc ${}^T z S z = {}^T x A x > 0$, et donc S est une matrice définie positive. \square

Corollaire 28.12 *La décomposition LU d'une matrice symétrique définie positive ne provoque jamais de division par 0.*

Démonstration : Soit A une matrice symétrique définie positive. Nous allons démontrer quelque chose de plus fort que l'énoncé du corollaire : tout pivot est strictement positif. Le premier pivot est a_{11} . Soit e_1 le premier vecteur unitaire, à partir duquel on obtient $a_{11} = {}^T e_1 A e_1 > 0$. Comme la première étape de la décomposition LU produit le complément de Schur de A par rapport à $A_1 = (a_{11})$, le lemme 28.11 implique que tous les pivots sont positifs par récurrence. \square

a) *Approximation par les moindres carrés*

Trouver une courbe approchée d'un ensemble de points donnés est une importante application des matrices symétriques définies positives. Supposons qu'on se donne un ensemble de m points du plan

$$(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m),$$

pour lesquels on sait que les y_i sont sujets à des erreurs de mesure. On voudrait déterminer une fonction $F(x)$ telle que

$$y_i = F(x_i) + \eta_i, \tag{28.31}$$

pour $i = 1, 2, \dots, m$, où les erreurs d'approximation η_i sont faibles. La forme de la fonction F dépend du problème posé. Ici, on suppose qu'elle a la forme d'une somme pondérée linéairement.

$$F(x) = \sum_{j=1}^n c_j f_j(x),$$

où le nombre de termes à sommer n et les **fonctions de base** f_j particulières sont choisies à partir de la connaissance qu'on a du problème. On prend souvent $f_j(x) = x^{j-1}$, ce qui signifie que

$$F(x) = c_1 + c_2 x + c_3 x^2 + \dots + c_n x^{n-1}$$

est un polynôme de degré $n - 1$ en x .

En prenant $n = m$, on peut calculer chaque y_i *exactement* dans l'équation (28.31). Toutefois, un F de degré élevé « approxime le bruit » en plus des données et donne généralement de mauvais résultats quand on s'en sert pour prédire la valeur de y pour des valeurs encore inconnues de x . En général, il vaut mieux choisir un n nettement inférieur à m et espérer que, en choisissant bien les coefficients c_j , on pourra obtenir une fonction F qui puisse trouver les schémas significatifs à partir des points de données sans être trop influencée par le bruit. Il existe certains principes théoriques pour le choix de n , mais cela dépasse le propos de ce livre. Dans tous les cas, une fois n

choisi, on aboutit à un ensemble d'équations surdéterminé dont on veut approximer la solution. Montrons comment on peut y parvenir. Soit

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix}$$

la matrice des valeurs des fonctions de base aux points donnés ; autrement dit, $a_{ij} = f_j(x_i)$. Soit $c = (c_k)$ le vecteur de coefficients de taille n désiré. Alors

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

est le vecteur de taille m des « valeurs prédites » de y . Donc,

$$\eta = Ac - y$$

est le vecteur de taille m des **erreurs d'approximation**.

Pour minimiser les erreurs d'approximation, on choisit de minimiser la norme du vecteur d'erreurs η , ce qui nous donne une **solution à moindres carrés**, puisque

$$\|\eta\| = \left(\sum_{i=1}^m \eta_i^2 \right)^{1/2}.$$

Comme

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij}c_j - y_i \right)^2,$$

on peut minimiser $\|\eta\|$ en dérivant $\|\eta\|^2$ par rapport à chaque c_k , puis en mettant le résultat à 0 :

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^m 2 \left(\sum_{j=1}^n a_{ij}c_j - y_i \right) a_{ik} = 0. \quad (28.32)$$

Les n équations (28.32) pour $k = 1, 2, \dots, n$ sont équivalentes à l'unique équation matricielle

$${}^T(Ac - y)A = 0$$

ou, si l'on préfère (en s'appuyant sur l'exercice 28.1.2), à

$${}^T A(Ac - y) = 0 ,$$

ce qui implique

$${}^T AAc = {}^T A y . \tag{28.33}$$

En statistiques, cette équation est dite *équation normale*. La matrice ${}^T AA$ est symétrique d'après l'exercice 28.1.2 ; et si A possède un rang colonne plein, alors d'après le théorème 28.6 ${}^T AA$ est également définie positive. Donc, $({}^T AA)^{-1}$ existe, et la solution à l'équation (28.33) est

$$\begin{aligned} c &= (({}^T AA)^{-1} {}^T A) y \\ &= A^+ y , \end{aligned} \tag{28.34}$$

où la matrice $A^+ = (({}^T AA)^{-1} {}^T A)$ est appelée *pseudo-inverse* de la matrice A . Le pseudo-inverse est une généralisation naturelle de la notion d'inverse au cas où A n'est pas carrée. (Comparer l'équation (28.34) comme solution approchée de $Ac = y$ et l'expression $A^{-1}b$ comme solution exacte de $Ax = b$.)

Prenons un exemple d'approximation par la méthode des moindres carrés. Supposons qu'on se donne les 5 points de données

$$\begin{aligned} (x_1, y_1) &= (-1, 2) , \\ (x_2, y_2) &= (1, 1) , \\ (x_3, y_3) &= (2, 1) , \\ (x_4, y_4) &= (3, 0) , \\ (x_5, y_5) &= (5, 3) , \end{aligned}$$

représentés par des points noirs sur la figure 28.3. On souhaite approximer ces points par un polynôme quadratique

$$F(x) = c_1 + c_2x + c_3x^2 .$$

On commence avec la matrice des valeurs des fonctions de base

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix} ,$$

dont le pseudo-inverse est

$$A^+ = \begin{pmatrix} 0,500 & 0,300 & 0,200 & 0,100 & -0,100 \\ -0,388 & 0,093 & 0,190 & 0,193 & -0,088 \\ 0,060 & -0,036 & -0,048 & -0,036 & 0,060 \end{pmatrix} .$$

En multipliant y par A^+ , on obtient le vecteur de coefficients

$$c = \begin{pmatrix} 1,200 \\ -0,757 \\ 0,214 \end{pmatrix},$$

ce qui correspond au polynôme quadratique

$$F(x) = 1,200 - 0,757x + 0,214x^2$$

comme interpolation optimale, au sens des moindres carrés, des données concernées.

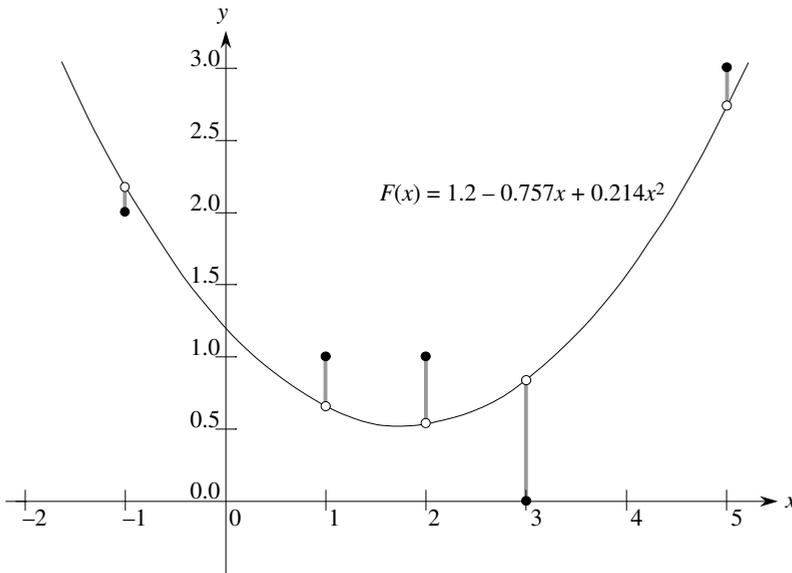


Figure 28.3 Approximation par moindres carrés, avec un polynôme quadratique, de l'ensemble des cinq points $\{(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)\}$. Les points en noir représentent les points de données, et les points en blanc sont leurs valeurs estimées prédites par le polynôme $F(x) = 1,2 - 0,757x + 0,214x^2$, polynôme quadratique qui minimise la somme des erreurs élevées au carré. L'erreur pour chaque point de données est représentée par une ligne ombrée.

D'un point de vue pratique, on résout l'équation normale (28.33) en multipliant y par ${}^T A$, puis en cherchant une décomposition LU de ${}^T A A$. Si A est de rang plein, on est assuré que la matrice ${}^T A A$ n'est pas singulière, puisqu'elle est symétrique et définie positive. (Voir exercice 28.1.2 et théorème 28.6.)

Exercices

28.5.1 Démontrer que tout élément diagonal d'une matrice symétrique et définie positive est positif.

28.5.2 Soit $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ une matrice 2×2 symétrique et définie positive. Démontrer que son déterminant $ac - b^2$ est positif en « complétant le carré » à la manière de la démonstration du lemme 28.11.

28.5.3 Démontrer que l'élément maximal d'une matrice symétrique et définie positive se trouve sur la diagonale.

28.5.4 Démontrer que le déterminant de chaque sous-matrice initiale d'une matrice symétrique définie positive est positif.

28.5.5 Soit A_k la k ème sous-matrice initiale d'une matrice symétrique définie positive A . Démontrer que $\det(A_k) / \det(A_{k-1})$ est le k ème pivot de la décomposition LU, où par convention $\det(A_0) = 1$.

28.5.6 Trouver la fonction de la forme

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

qui approche le mieux, au sens des moindres carrés, les points suivants :

$$(1, 1), (2, 1), (3, 3), (4, 8) .$$

28.5.7 Montrer que le pseudo inverse A^+ vérifie les quatre équations suivantes :

$$\begin{aligned} AA^+A &= A , \\ A^+AA^+ &= A^+ , \\ {}^T(AA^+) &= AA^+ , \\ {}^T(A^+A) &= A^+A . \end{aligned}$$

PROBLÈMES

28.1. Système tridiagonal d'équations linéaires

On considère la matrice tridiagonale

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} .$$

- Trouver une décomposition LU de A .
- Résoudre l'équation $Ax = {}^T (1 \ 1 \ 1 \ 1 \ 1)$ à l'aide de substitutions avant et arrière.

- c. Trouver l'inverse de A .
- d. Montrer que, pour toute matrice $n \times n$ symétrique, définie positive et tridiagonale A et pour tout vecteur b de dimension n , l'équation $Ax = b$ peut être résolue en temps $O(n)$ en effectuant une décomposition LU. Prouver que toute méthode basée sur le calcul de A^{-1} est asymptotiquement plus coûteuse dans le cas le plus défavorable.
- e. Montrer que, pour toute matrice $n \times n$ tridiagonale et non singulière A et pour tout vecteur b de dimension n , l'équation $Ax = b$ peut être résolue en temps $O(n)$ en effectuant une décomposition LUP.

28.2. Splines

Une méthode pratique d'interpolation d'un ensemble de points consiste à utiliser des *splines cubiques*. Soit un ensemble $\{(x_i, y_i) : i = 0, 1, \dots, n\}$ de $n + 1$ paires point-valeur, où $x_0 < x_1 < \dots < x_n$. On souhaite approximer l'ensemble des points par une courbe cubique par morceaux (spline) $f(x)$. En d'autres termes, la courbe $f(x)$ est constituée de n polynômes cubiques $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ pour $i = 0, 1, \dots, n-1$, tels que si x tombe dans l'intervalle $x_i \leq x \leq x_{i+1}$, alors la valeur de la courbe est donnée par $f(x) = f_i(x - x_i)$. Les points x_i de « raccordement » des polynômes cubiques s'appellent les *nœuds*. Par commodité, on supposera que $x_i = i$ pour $i = 0, 1, \dots, n$.

Pour assurer la continuité de f , on exige que

$$\begin{aligned} f(x_i) &= f_i(0) = y_i, \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

pour $i = 0, 1, \dots, n-1$. Pour assurer que f est suffisamment lissée, on veut aussi que la dérivée première soit continue en chaque nœud :

$$f'(x_{i+1}) = f'_i(1) = f'_{i+1}(0)$$

pour $i = 0, 1, \dots, n-1$.

- a. Supposons que pour $i = 0, 1, \dots, n$, on se donne non seulement les paires point-valeur $\{(x_i, y_i)\}$, mais aussi les dérivées premières $D_i = f'(x_i)$ en chaque nœud. Exprimer chaque coefficient a_i, b_i, c_i et d_i en fonction des valeurs y_i, y_{i+1}, D_i et D_{i+1} . (Rappelez-vous que $x_i = i$.) En combien de temps peut-on calculer les $4n$ coefficients à partir des paires point-valeur et des dérivées premières ?

Reste le problème du choix des dérivées premières de $f(x)$ en chaque nœud. Une méthode consiste à contraindre les dérivées secondes à être continues en les nœuds :

$$f''(x_{i+1}) = f''_i(1) = f''_{i+1}(0)$$

pour $i = 0, 1, \dots, n-1$. Au premier et au dernier nœud, on suppose que $f''(x_0) = f''_0(0) = 0$ et $f''(x_n) = f''_n(1) = 0$; ces hypothèses font de $f(x)$ une spline cubique *naturelle*.

- b. Utiliser les contraintes de continuité de la dérivée seconde pour montrer que pour $i = 1, 2, \dots, n - 1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}). \quad (28.35)$$

- c. Montrer que

$$2D_0 + D_1 = 3(y_1 - y_0), \quad (28.36)$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}). \quad (28.37)$$

- d. Réécrire les équations (28.35)–(28.37) sous forme d'équation matricielle utilisant le vecteur d'inconnues $D = \langle D_0, D_1, \dots, D_n \rangle$. Quels attributs la matrice de l'équation doit-elle posséder ?
- e. Montrer qu'un ensemble de $n + 1$ paires point-valeur peut être interpolé par une spline cubique naturelle en temps $O(n)$ (voir problème 28.5.8).
- f. Montrer comment déterminer une spline cubique naturelle qui interpole un ensemble de $n + 1$ points (x_i, y_i) vérifiant $x_0 < x_1 < \dots < x_n$, même quand x_i n'est pas forcément égal à i . Quelle équation matricielle faut-il résoudre, et combien de temps votre algorithme demande-t-il ?

NOTES

Il existe de nombreux textes excellents qui décrivent le calcul numérique et scientifique beaucoup plus en détail que dans ce livre. On pourra lire notamment : George et Liu [113], Golub et Van Loan [125], Press, Flannery, Teukolsky, et Vetterling [248, 249], et Strang [285, 286].

Golub et Van Loan [125] traitent de la stabilité numérique. Ils expliquent pourquoi $\det(A)$ n'est pas toujours un bon indicateur de la stabilité d'une matrice A , et ils proposent d'utiliser plutôt $\|A\|_\infty \|A^{-1}\|_\infty$, où $\|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$. Ils traitent aussi du problème consistant à calculer cette valeur sans calculer effectivement A^{-1} .

La publication de l'algorithme de Strassen en 1969 [287] a provoqué une grande excitation. Auparavant, on avait du mal à imaginer que l'algorithme naïf puisse être amélioré. La borne supérieure asymptotique, concernant la complexité de la multiplication matricielle, a été depuis lors considérablement améliorée. L'algorithme le plus efficace asymptotiquement connu à ce jour, pour la multiplication de matrices $n \times n$, est dû à Coppersmith et Winograd [70] ; il s'exécute en temps $O(n^{2,376})$. La représentation graphique de l'algorithme de Strassen est dû à Paterson [238].

La méthode du pivot de Gauss, sur laquelle sont fondées les décompositions LU et LUP, fut la première méthode systématique de résolution de systèmes d'équations linéaires. Elle est également connue comme l'un des tout premiers algorithmes numériques. Bien que cet algorithme ait été connu auparavant, on attribue généralement sa découverte à C. F. Gauss (1777–1855). Dans son fameux article, [287], Strassen montra également qu'une matrice $n \times n$ pouvait être inversée en temps $O(n^{\lg 7})$. Winograd [317] fut le premier à montrer que la multiplication matricielle n'était pas plus complexe que l'inversion de matrice, et la réciproque est due à Aho, Hopcroft et Ullman [5].

Une autre décomposition matricielle majeure est la *décomposition en valeurs singulières*, appelée aussi *SVD*. Avec la SVD, une matrice $m \times n$ A est factorisée en $A = Q_1 \Sigma Q_2^T$, où Σ est une matrice $m \times n$ n'ayant de valeurs non nulles que sur la diagonale, Q_1 est une matrice $m \times m$ ayant des colonnes mutuellement orthonormales et Q_2 est une matrice $n \times n$ ayant aussi des colonnes mutuellement orthonormales. Deux vecteurs sont *orthonormaux* si leur produit scalaire est 0 et si chaque vecteur a pour norme 1. Les livres de Strang [285, 286] et de Golub et Van Loan [125] contiennent de bons exposés de la méthode SVD.

Strang [286] contient une excellente présentation des matrices symétriques définies positives et de l'algèbre linéaire en général.

Chapitre 29

Programmation linéaire

Nombreux sont les problèmes susceptibles d'être formulés en tant que maximisation ou minimisation d'un objectif en fonction de ressources limitées et de contraintes mutuellement rivales. Si l'on arrive à exprimer l'objectif sous la forme d'une fonction linéaire de certaines variables et que l'on peut spécifier les contraintes concernant les ressources sous la forme d'égalités ou d'inégalités sur ces variables, alors on a un ***problème de programmation linéaire***. Les programmes linéaires surgissent dans une foule d'applications concrètes. Nous commencerons par l'étude d'une application électorale.

a) Un problème tiré de la politique

Supposez que vous soyez candidat à la députation. Votre circonscription comprend une partie urbaine, des banlieues et une partie rurale. Ces trois zones contiennent respectivement 100 000, 200 000 et 50 000 électeurs inscrits. Pour pouvoir appliquer votre programme, vous voulez gagner la majorité des voix dans chacune des trois zones. Vous avez compris, cependant, que gagner des électeurs peut se révéler plus ou moins intéressant pour l'avenir selon l'endroit où vous obtenez les suffrages. Les grands axes de votre programme sont la construction de routes, le renforcement de la lutte anti-drogue, les subventions aux agriculteurs et une taxe sur les engrais destinée à améliorer la qualité de l'eau. Les enquêtes menées par votre équipe vous permettent d'estimer le nombre de votes que vous perdriez ou gagneriez dans chaque catégorie de population en dépensant 1000€ de publicité sur chaque problème. Ces informations sont données dans le tableau de la figure 29.1. Dans ce tableau, chaque donnée individuelle décrit le nombre de milliers d'électeurs de la ville, des banlieues ou de

la campagne que vous pourriez gagner en dépensant 1000€ de publicité en faveur d'une question particulière. Les valeurs négatives représentent les suffrages que vous pourriez perdre. Votre travail consiste à calculer l'investissement minimal que vous devez dépenser pour gagner 50 000 votes urbains, 100 000 votes de banlieusards et 25 000 votes ruraux.

stratégie	ville	banlieues	campagne
construction de routes	-2	5	3
lutte anti-drogue	8	2	-5
subventions aux agriculteurs	0	0	10
taxe sur les engrais	10	0	-2

Figure 29.1 Les effets des politiques sur les électeurs. Chaque donnée individuelle représente le nombre de milliers d'électeurs de la ville, des banlieues ou de la campagne que vous pourriez gagner en dépensant 1000€ de publicité en faveur d'une politique concernant une question particulière. Les valeurs négatives représentent les suffrages que vous perdriez.

En essayant successivement diverses combinaisons, vous finiriez peut-être par trouver une stratégie permettant de gagner le nombre de suffrages requis, mais cette stratégie risque fort de ne pas être la plus économique. Supposons, par exemple, que vous consacriez 20 000 € à la publicité en faveur de la construction de routes, 0 € à la publicité en faveur de la lutte anti-drogue, 4 000 € à la publicité en faveur des subventions aux agriculteurs et 9 000 € à la publicité en faveur de la taxe sur les engrais. En pareil cas, vous gagneriez $20(-2) + 0(8) + 4(0) + 9(10) = 50$ mille votes urbains, $20(5) + 0(2) + 4(0) + 9(0) = 100$ mille votes de banlieusards et $20(3) + 0(-5) + 4(10) + 9(-2) = 82$ mille votes ruraux. Vous obtiendriez le nombre exact de suffrages requis en ville et dans les banlieues, et plus de suffrages qu'il n'en faut dans la campagne. (En fait, dans la portion rurale, vous obtiendriez plus de suffrages qu'il n'y a d'électeurs !) Pour gagner tous ces votes, vous auriez dépensé $20 + 0 + 4 + 9 = 33$ mille euros de publicité.

On peut se demander, naturellement, si cette stratégie est la meilleure : peut-on arriver au même résultat en dépensant moins ? Vous pourriez procéder à d'autres essais au petit bonheur la chance, mais mieux vaut disposer d'une méthode systématique pour répondre à ce genre de question. Pour ce faire, nous allons exprimer le problème sous forme mathématique. Nous définirons 4 variables :

- x_1 est le nombre de milliers d'euros dépensés en publicité pour la construction de routes,
- x_2 est le nombre de milliers d'euros dépensés en publicité pour la lutte anti-drogue,
- x_3 est le nombre de milliers d'euros dépensés en publicité pour les subventions aux agriculteurs et
- x_4 est le nombre de milliers d'euros dépensés en publicité pour la taxe sur les engrais.

L'objectif de gagner au moins 50 000 votes urbains peut s'exprimer sous la forme suivante

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 . \tag{29.1}$$

De la même façon, nous pouvons formuler comme suit les objectifs de gagner au moins 100 000 votes de banlieusards et 25 000 votes ruraux

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

et

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 . \tag{29.3}$$

Toute configuration des variables x_1, x_2, x_3, x_4 qui satisfait aux inégalités (29.1)–(29.3) constitue une stratégie permettant de gagner un nombre suffisant d'électeurs de chaque zone. Pour réduire les frais au maximum, on aimerait minimiser les dépenses de publicité. On voudrait donc minimiser l'expression

$$x_1 + x_2 + x_3 + x_4 . \tag{29.4}$$

Comme il n'existe pas de publicité à coût négatif, il faut en outre que

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, \text{ et } x_4 \geq 0 . \tag{29.5}$$

En combinant les inégalités (29.1)–(29.3) et (29.5) avec l'objectif de minimiser (29.4), on obtient ce que l'on appelle un « programme linéaire ». Nous représentons ce problème ainsi

minimiser $x_1 + x_2 + x_3 + x_4$ (29.6)

sous les contraintes

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \tag{29.7}$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.8}$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \tag{29.9}$$

$$x_1, x_2, x_3, x_4 \geq 0 . \tag{29.10}$$

La solution de ce programme linéaire fournira une stratégie optimale de campagne politique.

b) Programmes linéaires généraux

Dans le cas général, un problème de programmation linéaire consiste à optimiser une fonction linéaire soumise à un ensemble d'inégalités linéaires. Étant donné un ensemble de nombres réels a_1, a_2, \dots, a_n et un ensemble de variables x_1, x_2, \dots, x_n , on définit une **fonction linéaire** f de ces variables par

$$f(x_1, x_2, \dots, x_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n = \sum_{j=1}^n a_jx_j .$$

Si b est un nombre réel et f une fonction linéaire, alors l'équation

$$f(x_1, x_2, \dots, x_n) = b$$

est une *égalité linéaire* et les inégalités

$$f(x_1, x_2, \dots, x_n) \leq b$$

et

$$f(x_1, x_2, \dots, x_n) \geq b$$

sont des *inégalités linéaires*. Le terme *contraintes linéaires* représente des égalités ou des inégalités linéaires. Dans la programmation linéaire, les inégalités strictes sont interdites. De manière formelle, un *problème de programmation linéaire* consiste à minimiser ou maximiser une fonction linéaire soumise à un ensemble fini de contraintes linéaires. Si l'objectif est de minimiser, alors le programme linéaire porte le nom de *programme linéaire de minimisation* ; si l'objectif est de maximiser, alors le programme linéaire porte le nom de *programme linéaire de maximisation*.

Le reste du chapitre couvrira la formulation et la résolution des programmes linéaires. Il existe plusieurs algorithmes à temps polynomial pour la programmation linéaire, mais nous ne les étudierons pas dans ce chapitre. À la place, nous verrons l'algorithme du simplexe qui est le plus vieil algorithme de programmation linéaire. Cet algorithme ne s'exécute pas en temps polynomial dans le cas le plus défavorable, mais il est relativement efficace et largement utilisé en pratique.

c) Aperçu de la programmation linéaire

Pour décrire propriétés et algorithmes des programmes linéaires, il est commode d'avoir des formats permettant de les exprimer. Dans ce chapitre nous emploierons deux formes, *forme canonique* et *forme standard*. Nous les définirons de manière plus précise à la section 29.1. De manière informelle, un programme linéaire sous forme canonique est la maximisation d'une fonction linéaire soumise à *inégalités* linéaires, alors qu'un programme linéaire sous forme standard est la maximisation d'une fonction linéaire soumise à des *égalités* linéaires. En principe, nous utiliserons la forme canonique pour exprimer les programmes linéaires, mais il est plus commode d'utiliser la forme standard pour décrire les détails de l'algorithme du simplexe. Pour l'instant, nous restreindrons notre étude à la maximisation d'une fonction linéaire de n variables soumise à un ensemble de m inégalités linéaires.

Regardons tout d'abord le programme linéaire suivant, à deux variables :

$$\text{maximiser} \quad x_1 + x_2 \quad (29.11)$$

sous les contraintes

$$4x_1 - x_2 \leq 8 \quad (29.12)$$

$$2x_1 + x_2 \leq 10 \quad (29.13)$$

$$5x_1 - 2x_2 \geq -2 \quad (29.14)$$

$$x_1, x_2 \geq 0 \quad (29.15)$$

Toute configuration des variables x_1 et x_2 satisfaisant à toutes les contraintes (29.12)–(29.15) est dite **solution réalisable** du programme linéaire. Si nous traçons une représentation graphique des contraintes dans le système de coordonnées cartésiennes (x_1, x_2) , comme sur la figure 29.2(a), nous voyons que l'ensemble des solutions réalisables (colorié en gris sur la figure) forme une région convexe⁽¹⁾ du plan. Cette région convexe porte le nom de **région de réalisabilité**. La fonction que nous voulons maximiser est dite **fonction objectif**. Théoriquement, on pourrait évaluer la fonction objectif $x_1 + x_2$ en chaque point de la région de réalisabilité ; la valeur de la fonction objectif en un point particulier est dite **valeur de l'objectif**. On pourrait ensuite identifier comme solution optimale un point maximisant la valeur objective. Pour cet exemple (et pour la plupart des programmes linéaires), la région de réalisabilité renferme un nombre infini de points ; nous voulons donc trouver un moyen efficace de déterminer un point maximisant la valeur de l'objectif sans avoir à évaluer explicitement la fonction objectif en chaque point de la région de réalisabilité.

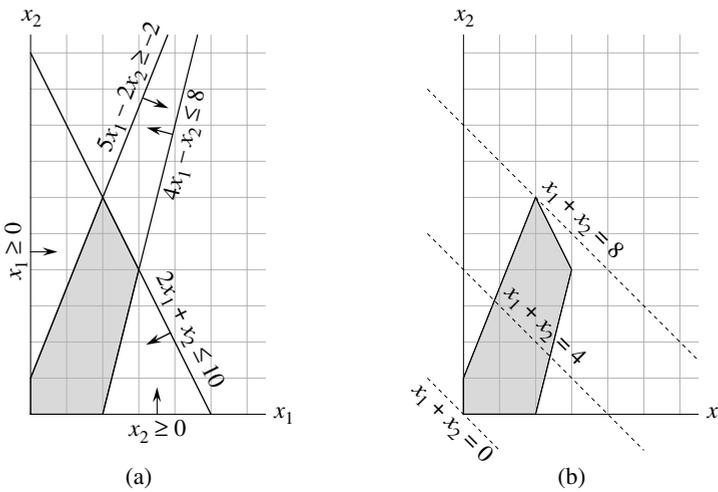


Figure 29.2 (a) Le programme linéaire des lignes (29.12)–(29.15). Chaque contrainte est représentée par une droite et une direction. L'intersection des contraintes, qui est la région de réalisabilité, est coloriée en gris. (b) Les lignes en pointillés montrent respectivement les points pour lesquels la valeur objective est 0, 4 et 8. La solution optimale du programme linéaire est $x_1 = 2$ et $x_2 = 6$ avec la valeur objective 8.

Avec deux dimensions, nous pouvons optimiser *via* une procédure graphique. L'ensemble des points pour lesquels $x_1 + x_2 = z$, pour tout z , est une droite de pente -1 . Si nous représentons graphiquement $x_1 + x_2 = 0$, nous obtenons la droite de pente -1 passant par l'origine, comme sur la figure 29.2(b). L'intersection de cette

(1) Une définition intuitive d'une région convexe est que, pour tout couple de points de la région, le segment reliant ces deux points est inclus dans la région

droite et de la région de réalisabilité est l'ensemble des solutions réalisables ayant la valeur objective 0. Ici, cette intersection de la droite et de la région de réalisabilité est le point $(0, 0)$. Plus généralement, pour tout z , l'intersection de la droite $x_1 + x_2 = z$ et de la région de réalisabilité est l'ensemble des solutions réalisables ayant comme valeur de l'objectif z . La figure 29.2(b) montre les droites $x_1 + x_2 = 0$, $x_1 + x_2 = 4$ et $x_1 + x_2 = 8$. Comme la région de réalisabilité de la figure 29.2 est bornée, il doit exister une certaine valeur maximale z pour laquelle l'intersection de la droite $x_1 + x_2 = z$ et de la région de réalisabilité n'est pas vide. Tout point auquel cela se produit est une solution optimale du programme linéaire; ici, c'est le point de coordonnées $x_1 = 2$ et $x_2 = 6$ avec la valeur de l'objectif 8. Ce n'est pas le fait du hasard qu'une solution optimale du programme linéaire se situe en un sommet de la région de réalisabilité. La valeur maximale de z pour laquelle la droite $x_1 + x_2 = z$ coupe la région de réalisabilité doit être sur la frontière de la région de réalisabilité; par conséquent, l'intersection de cette droite et de la frontière de la région de réalisabilité est soit un sommet, soit un segment de droite. Si l'intersection est un sommet, alors il n'y a qu'une seule solution optimale et c'est ce sommet. Si l'intersection est un segment de droite, chaque point de ce segment doit avoir la même valeur de l'objectif; en particulier, les deux extrémités du segment sont des solutions optimales. Comme chaque extrémité d'un segment est un sommet, dans ce cas aussi il y a une solution optimale au niveau d'un sommet.

Il n'est pas facile de représenter graphiquement les programmes linéaires de plus de deux variables, mais la même intuition reste de mise. Avec trois variables, chaque contrainte est décrite par un demi-espace de l'espace à trois dimensions. L'intersection de ces demi-espaces forme la région de réalisabilité. L'ensemble des points pour lesquels la fonction objectif vaut z est désormais un plan. Si tous les coefficients de la fonction objectif sont positifs et si l'origine est une solution réalisable du programme linéaire, alors, quand nous éloignons ce plan de l'origine, nous trouvons des points ayant des valeurs de l'objectif croissantes. (Si l'origine n'est pas réalisable ou si certains coefficients de la fonction objectif sont négatifs, la représentation intuitive se complique quelque peu.) Comme dans le cas à deux dimensions, du fait que la région de réalisabilité est convexe, l'ensemble des points maximisant la valeur de l'objectif doit inclure un sommet de la région de réalisabilité. De même, si nous avons n variables, chaque contrainte définit un demi-espace de l'espace à n dimensions. La région de réalisabilité formée par l'intersection de ces demi-espaces s'appelle un *simplexe*. La fonction objectif est maintenant un hyperplan et, compte tenu de la convexité, une solution optimale se situe, ici aussi, en un sommet du simplexe.

L'*algorithme du simplexe* prend en entrée un programme linéaire et donne en sortie une solution optimale. Il commence en un certain sommet du simplexe, puis effectue une suite d'itérations. À chaque itération, l'algorithme emprunte un côté du simplexe pour passer du sommet courant à un sommet adjacent dont la valeur objectif n'est pas inférieure à celle du sommet actuel (en général, elle est strictement

supérieure.) L'algorithme du simplexe se termine quand il atteint un maximum local, lequel est un sommet ayant une valeur objectif supérieure à celles de tous les sommets adjacents. Comme la région de réalisabilité est convexe et que la fonction objectif est linéaire, cet optimum local est en fait un optimum global. À la section 29.4, nous utiliserons un concept appelé « dualité » pour montrer que la solution produite par l'algorithme du simplexe est bien optimale.

Bien que la vision géométrique fournisse une bonne vue intuitive du fonctionnement de l'algorithme du simplexe, nous ne nous y référerons pas explicitement quand nous exposerons les détails de l'algorithme à la section 29.3. Nous prendrons plutôt une vision algébrique. Nous commencerons par écrire le programme linéaire sous forme standard, laquelle est un ensemble d'égalités linéaires. Ces égalités linéaires expriment certaines des variables, dites « variables hors-base », à partir d'autres variables, dites « variables hors-base ». Le passage d'un sommet à un autre se fait *via* transformation d'une variable de base en une variable hors-base et transformation d'une variable hors-base en variable de base. Cette opération porte le nom de « pivot » et, vue de manière algébrique, ce n'est rien d'autre que la réécriture du programme linéaire sous une forme standard équivalente.

L'exemple à deux variables précédemment décrit était des plus simples. Nous devons traiter plusieurs autres détails dans ce chapitre. Ces questions concernent l'identification des programmes linéaires dépourvus de solutions, des programmes linéaires dépourvus de solution optimale finie et des programmes linéaires pour lesquels l'origine n'est pas une solution réalisable.

d) Applications de la programmation linéaire

La programmation linéaire a un grand nombre d'applications. N'importe quel traité de recherche opérationnelle est rempli d'exemples de programmation linéaire, et c'est un outil standard que l'on enseigne dans la plupart des écoles de gestion. Le scénario électoral est un exemple classique. Voici deux autres exemples de programmation linéaire :

- Une compagnie aérienne veut planifier l'affectation des équipages. Les réglementations aériennes imposent moult contraintes, par exemple le fait qu'un navigant ne peut pas travailler plus de tant d'heures d'affilée. Or, la compagnie veut affecter les équipages aux vols de façon à minimiser le volume de personnel employé.
- Une compagnie pétrolière veut savoir où elle doit faire des forages. Faire un forage coûte pas mal d'argent et, selon la nature du terrain, la rentabilité de l'opération n'est pas systématiquement garantie. La compagnie dispose d'un budget limité en matière de recherche de nouveaux sites de forage et, avec ce budget, elle veut maximiser la quantité de pétrole qu'elle espère trouver.

Les programmes linéaires sont aussi très utiles pour modéliser et résoudre des problèmes de graphe et des problèmes combinatoires, tels que ceux donnés dans ce livre. Nous avons déjà vu un cas particulier de programmation linéaire utilisé pour résoudre

des systèmes de contraintes de potentiel à la section 24.4. À la section 29.2, nous apprendrons à exprimer sous la forme de programmes linéaires plusieurs problèmes concernant les graphes et les flots dans les réseaux de transport. À la section 35.4, nous utiliserons la programmation linéaire pour trouver une solution approchée à un autre problème de graphe.

e) Algorithmes de programmation linéaire

Ce chapitre traite de l'algorithme du simplexe. Cet algorithme, quand on l'implémente soigneusement, permet souvent de résoudre des programmes linéaires généraux avec des temps performants. Néanmoins, en cherchant bien, l'on trouve des cas de figure où cet algorithme demande des temps exponentiels. Le premier algorithme à temps polynomial de programmation linéaire fut l'*algorithme de l'ellipsoïde* qui, en pratique, tourne lentement. Une autre classe d'algorithmes à temps polynomial porte le nom *deméthodes de point intérieur*. Par rapport à l'algorithme du simplexe, qui se déplace à l'extérieur de la région de réalisabilité en géant une solution réalisable qui est un sommet du simplexe à chaque itération, ces algorithmes se déplacent à l'intérieur de la région de réalisabilité. Les solutions intermédiaires, bien que réalisables, ne sont pas forcément des sommets du simplexe, mais la solution finale est un sommet. Le premier algorithme de ce genre fut inventé par Karmarkar. Pour les entrées volumineuses, les performances des algorithmes à point intérieur peuvent être comparables, voire supérieures, à celles de l'algorithme du simplexe.

Si nous ajoutons à un programme linéaire la contrainte supplémentaire que toutes les variables prennent des valeurs entières, alors nous avons un *programme linéaire entier*. L'exercice 34.5.3 vous demandera de montrer que le simple fait de trouver une solution réalisable à ce problème est NP-complet ; comme l'on ne connaît aucun algorithme à temps polynomial pour quelque problème NP-complet que ce soit, il n'existe donc pas d'algorithme connu à temps polynomial pour la programmation linéaire entière. En comparaison, un problème général de programmation linéaire est résoluble en temps polynomial.

Dans ce chapitre, si nous avons un programme linéaire avec les variables $x = (x_1, x_2, \dots, x_n)$ et que nous voulons nous référer à une configuration particulière des variables, nous emploierons la notation $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$.

29.1 FORME CANONIQUE ET FORME STANDARD

Cette section va présenter deux formats, à savoir la forme canonique et la forme standard (slack), qui sont utiles pour spécifier et utiliser des programmes linéaires. Avec la forme canonique, toutes les contraintes sont des inégalités, alors qu'avec la forme standard ce sont des égalités.

f) Forme canonique

Avec la **forme canonique**, nous avons n nombres réels c_1, c_2, \dots, c_n ; m nombres réels b_1, b_2, \dots, b_m ; et mn nombres réels a_{ij} pour $i = 1, 2, \dots, m$ et $j = 1, 2, \dots, n$. Nous voulons trouver n nombres réels x_1, x_2, \dots, x_n qui

$$\text{maximisent} \quad \sum_{j=1}^n c_j x_j \quad (29.16)$$

en tenant compte des contraintes

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \quad \text{pour } i = 1, 2, \dots, m \quad (29.17)$$

$$x_j \geq 0 \quad \text{pour } j = 1, 2, \dots, n. \quad (29.18)$$

En généralisant la terminologie introduite avec le programme linéaire à deux variables, nous appelons l'expression (29.16) la **fonction objectif** et les $n + m$ inégalités des lignes (29.17) et (29.18) les **contraintes**. Les n contraintes de la ligne (29.18) sont dites **contraintes de positivité**. Un programme linéaire quelconque n'est pas obligé d'avoir des contraintes de positivité, mais il en faut pour la forme canonique. Il est parfois commode d'exprimer un programme linéaire sous une forme plus compacte. Si nous créons une matrice $m \times n$ appelée ici $A = (a_{ij})$, un vecteur à m dimensions $b = (b_i)$, un vecteur à n dimensions $c = (c_j)$ et un vecteur à n dimensions $x = (x_j)$, alors nous pouvons réécrire le programme linéaire de (29.16)–(29.18) sous la forme

$$\text{maximiser} \quad c^T x \quad (29.19)$$

sous les des contraintes

$$Ax \leq b \quad (29.20)$$

$$x \geq 0. \quad (29.21)$$

Sur la ligne (29.19), $c^T x$ est le produit scalaire de deux vecteurs. Sur la ligne (29.20), Ax est un produit matrice-vecteur. Sur la ligne (29.21), $x \geq 0$ signifie que chaque composante du vecteur x doit être positive. On voit que l'on peut exprimer un programme linéaire sous forme canonique à l'aide d'un tuple (A, b, c) , et l'on adoptera la convention que A , b et c ont toujours les dimensions précédemment spécifiées.

Nous allons maintenant introduire la terminologie employée pour décrire les solutions de programmes linéaires. Une partie de cette terminologie a déjà été vue dans l'exemple précédent de programme linéaire à deux variables. Nous appellerons **solution réalisable** toute configuration des variables \bar{x} qui satisfait à toutes les contraintes; une configuration des variables \bar{x} qui ne satisfait pas à au moins une contrainte est dite **solution irréalisable**. Nous dirons qu'une solution \bar{x} a la **valeur de l'objectif** $c^T \bar{x}$. Une solution réalisable \bar{x} dont la valeur objective est supérieure à toutes les solutions réalisables est une **solution optimale**, et nous dirons que sa valeur de l'objectif $c^T \bar{x}$

est la *valeur de l'objectif optimale*. Si un programme linéaire n'a aucune solution réalisable, nous dirons qu'il est *irréalisable* ; autrement, il est *réalisable*. Si un programme linéaire a des solutions réalisable sans toutefois avoir de valeur de l'objectif optimale finie, nous dirons qu'il est *non borné*. L'exercice 29.1.9 vous demandera de montrer qu'un programme linéaire peut avoir une valeur de l'objectif optimale finie, même si la région de réalisabilité n'est pas bornée.

g) Conversion de programmes linéaires sous forme canonique

Il est toujours possible de convertir sous forme canonique un programme linéaire, donné en tant que minimisation ou maximisation d'une fonction linéaire soumise à des contraintes linéaires. Un programme linéaire peut ne pas être sous forme canonique, et ce pour l'une des quatre raisons suivantes :

- 1) La fonction objectif peut être une minimisation au lieu d'une maximisation.
- 2) Il peut y avoir des variables sans contraintes de positivité.
- 3) Il peut y avoir des *contraintes d'égalité*, lesquelles ont un signe d'égalité plutôt qu'un signe inférieur ou égal.
- 4) Il peut y avoir des *contraintes d'inégalité* qui ont un signe supérieur ou égal au lieu d'un signe inférieur ou égal.

Quand on convertit un programme linéaire L en un programme linéaire L' , on voudrait qu'une solution optimale pour L' donne une solution optimale pour L . Pour exprimer cette idée, nous dirons que deux programmes linéaires de maximisation L et L' sont des *équivalent* si, pour toute solution réalisable \bar{x} de L ayant pour valeur de l'objectif z , il existe une solution réalisable homologue \bar{x}' de L' ayant pour valeur de l'objectif z , et si, pour toute solution réalisable \bar{x}' de L' ayant pour valeur de l'objectif z , il existe une solution réalisable homologue \bar{x} de L ayant pour valeur de l'objectif z . (Cette définition n'implique pas de bijection entre solutions réalisables.) Un programme linéaire de minimisation L et un programme linéaire de maximisation L' sont équivalents si, pour toute solution réalisable \bar{x} de L ayant pour valeur de l'objectif z , il existe une solution réalisable homologue \bar{x}' de L' ayant pour valeur de l'objectif $-z$, et si, pour toute solution réalisable \bar{x}' de L' ayant pour valeur de l'objectif z , il existe une solution réalisable homologue \bar{x} de L ayant pour valeur de l'objectif $-z$.

Nous allons montrer comment supprimer, un par un, tous les problèmes potentiels précédemment énumérés. Une fois supprimé le problème, nous prouverons que le nouveau programme linéaire est équivalent à l'ancien.

Pour convertir un programme linéaire de minimisation L en un programme linéaire de maximisation L' équivalent, il suffit de prendre les opposés des coefficients dans la fonction objectif. Comme L et L' ont des ensembles identiques de solutions réalisables et que, pour toute solution réalisable, la valeur de l'objectif dans L est l'opposée de la valeur de l'objectif dans L' , les deux programmes linéaires sont équivalents.

Par exemple, si nous avons le programme linéaire

$$\text{minimiser} \quad -2x_1 + 3x_2$$

sous les contraintes

$$\begin{aligned} x_1 + x_2 &= 7 \\ x_1 - 2x_2 &\leq 4 \\ x_1 &\geq 0, \end{aligned}$$

et si nous prenons les opposés des coefficients de la fonction objectif, nous obtenons

$$\text{maximiser} \quad 2x_1 - 3x_2$$

sous les contraintes

$$\begin{aligned} x_1 + x_2 &= 7 \\ x_1 - 2x_2 &\leq 4 \\ x_1 &\geq 0. \end{aligned}$$

Montrons ensuite comment convertir un programme linéaire dans lequel certaines variables n'ont pas de contraintes de positivité en un programme dans lequel chaque variable a une contrainte de positivité. Supposez qu'une certaine variable x_j n'ait pas de contrainte de positivité. On remplace alors chaque occurrence de x_j par $x'_j - x''_j$, et l'on ajoute les contraintes de positivité $x'_j \geq 0$ et $x''_j \geq 0$. Ainsi donc, si la fonction objectif a un terme $c_j x_j$, il est remplacé par $c_j x'_j - c_j x''_j$, et si la contrainte i a un terme $a_{ij} x_j$, il est remplacé par $a_{ij} x'_j - a_{ij} x''_j$. Toute solution réalisable \bar{x} du nouveau programme linéaire correspond à une solution réalisable du programme originel avec $\bar{x}_j = \bar{x}'_j - \bar{x}''_j$ et avec la même valeur de l'objectif, ce qui entraîne que les deux programmes linéaires sont équivalents. On applique ce schéma de conversion à chaque variable dépourvue de contrainte de positivité pour produire un programme linéaire équivalent dans lequel toutes les variables ont des contraintes de positivité.

Continuant l'exemple, nous voulons faire en sorte que chaque variable ait une contrainte de positivité associée. La variable x_1 a une telle contrainte, mais pas la variable x_2 . Nous remplaçons donc x_2 par deux variables x'_2 et x''_2 , puis nous modifions le programme linéaire pour obtenir

$$\text{maximiser} \quad 2x_1 - 3x'_2 + 3x''_2$$

sous les contraintes

$$\begin{aligned} x_1 + x'_2 - x''_2 &= 7 \\ x_1 - 2x'_2 + 2x''_2 &\leq 4 \\ x_1, x'_2, x''_2 &\geq 0. \end{aligned} \tag{29.22}$$

Ensuite, nous convertissons les contraintes d'égalité en contraintes d'inégalité. Supposez qu'un programme linéaire ait une contrainte d'égalité $f(x_1, x_2, \dots, x_n) = b$. Comme $x = y$ si et seulement si l'on a à la fois $x \geq y$ et $x \leq y$, nous pouvons remplacer cette contrainte d'égalité par la paire de contraintes d'inégalité $f(x_1, x_2, \dots, x_n) \leq b$

et $f(x_1, x_2, \dots, x_n) \geq b$. En répétant cette conversion pour chaque contrainte d'égalité, nous obtenons un programme linéaire dans lequel toutes les contraintes sont des inégalités.

Enfin, nous pouvons convertir les contraintes supérieur ou égal en contraintes inférieur ou égal, en multipliant ces contraintes par -1 . C'est-à-dire que toute inégalité de la forme

$$\sum_{j=1}^n a_{ij}x_j \geq b_i$$

équivaut à

$$\sum_{j=1}^n -a_{ij}x_j \leq -b_i.$$

Ainsi, en remplaçant chaque coefficient a_{ij} par $-a_{ij}$ et chaque valeur b_i par $-b_i$, l'on obtient une contrainte inférieur ou égal équivalente.

Terminant notre exemple, nous remplaçons l'égalité de la contrainte (29.22) par deux inégalités, ce qui donne

maximiser

$$2x_1 - 3x'_2 + 3x''_2$$

en tenant compte des contraintes

$$x_1 + x'_2 - x''_2 \leq 7$$

$$x_1 + x'_2 - x''_2 \geq 7 \quad (29.23)$$

$$x_1 - 2x'_2 + 2x''_2 \leq 4$$

$$x_1, x'_2, x''_2 \geq 0.$$

Pour finir, nous prenons l'opposée de la contrainte (29.23). Pour rester cohérent au niveau des noms de variable, nous renommons x'_2 en x_2 et x''_2 en x_3 , arrivant ainsi à la forme canonique

maximiser

$$2x_1 - 3x_2 + 3x_3 \quad (29.24)$$

sous les contraintes

$$x_1 + x_2 - x_3 \leq 7 \quad (29.25)$$

$$-x_1 - x_2 + x_3 \leq -7 \quad (29.26)$$

$$x_1 - 2x_2 + 2x_3 \leq 4 \quad (29.27)$$

$$x_1, x_2, x_3 \geq 0. \quad (29.28)$$

h) Conversion de programmes linéaires sous forme standard

Pour résoudre efficacement un programme linéaire *via* l'algorithme du simplexe, on préfère l'exprimer sous une forme dans laquelle certaines des contraintes sont des contraintes d'égalité. Plus précisément, on le convertit en une forme dans laquelle

les contraintes de positivité sont les seules contraintes d'inégalité, toutes les autres contraintes étant des égalités. Soit

$$\sum_{j=1}^n a_{ij}x_j \leq b_i \quad (29.29)$$

une contrainte d'inégalité. Introduisons une nouvelle variable s et réécrivons l'inégalité (29.29) sous la forme des deux contraintes

$$s = b_i - \sum_{j=1}^n a_{ij}x_j, \quad (29.30)$$

$$s \geq 0. \quad (29.31)$$

s est dite **variable d'écart**, car elle mesure l'**écart**, ou différence, entre le côté gauche et le côté droit de l'équation (29.29). Comme l'inégalité (29.29) est vraie si et seulement si l'équation (29.30) et l'inégalité (29.31) sont vraies toutes les deux, on peut appliquer cette conversion à chaque contrainte d'inégalité d'un programme linéaire pour obtenir un programme linéaire équivalent dans lequel les seules contraintes d'inégalité sont les contraintes de positivité. Quand on convertit de la forme canonique vers la forme standard, on utilise x_{n+i} (au lieu de s) pour noter la variable d'écart associée à la i -ème inégalité. La i -ème contrainte est donc

$$x_{n+i} = b_i - \sum_{j=1}^n a_{ij}x_j, \quad (29.32)$$

accompagnée de la contrainte de positivité $x_{n+i} \geq 0$.

En appliquant cette conversion à chaque contrainte d'un programme linéaire sous forme canonique, on obtient un programme linéaire sous une autre forme. Par exemple, pour le programme linéaire décrit dans (29.24)–(29.28), nous introduisons des variables d'écart x_4 , x_5 et x_6 pour obtenir

$$\text{maximiser} \quad 2x_1 - 3x_2 + 3x_3 \quad (29.33)$$

sous les contraintes

$$x_4 = 7 - x_1 - x_2 + x_3 \quad (29.34)$$

$$x_5 = -7 + x_1 + x_2 - x_3 \quad (29.35)$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \quad (29.36)$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0. \quad (29.37)$$

Dans ce programme linéaire, toutes les contraintes, hormis les contraintes de positivité, sont des égalités et chaque variable est soumise à une contrainte de positivité. Nous écrivons chaque contrainte d'égalité avec l'une des variables dans le membre de gauche de l'égalité et les autres variables dans le membre de droite. En outre, chaque équation a le même ensemble de variables dans le membre de droite, et ces variables sont aussi les seules à figurer dans la fonction objectif. Les variables du membre de

gauche des égalités sont dites *variables de base*, alors que celles du membre de droite sont dites *variables hors-base*.

Pour les programmes linéaires satisfaisant à ces conditions, nous omettrons parfois les expressions « maximiser » et « sous les contraintes », et nous omettrons les contraintes de positivité explicites. Nous utiliserons en outre la variable z pour noter la valeur de la fonction objectif. Le format résultant porte le nom de *forme standard*. Si nous écrivons le programme linéaire donné dans (29.33)–(29.37) sous forme standard, nous obtenons

$$z = 2x_1 - 3x_2 + 3x_3 \quad (29.38)$$

$$x_4 = 7 - x_1 - x_2 + x_3 \quad (29.39)$$

$$x_5 = -7 + x_1 + x_2 - x_3 \quad (29.40)$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \quad (29.41)$$

Comme avec la forme canonique, il est commode d'avoir une notation plus compacte pour décrire une forme standard. Comme nous le verrons à la section 29.3, l'ensemble des variables de base et celui des variables hors-base changent au fur et à mesure de l'exécution de l'algorithme du simplexe. Nous utilisons N pour désigner l'ensemble des indices des variables hors-base et B pour désigner l'ensemble des indices des variables de base. Nous avons toujours $|N| = n$, $|B| = m$ et $N \cup B = \{1, 2, \dots, n + m\}$. Les équations sont indicées par les valeurs contenues dans B , et les variables des côtés droits sont indicées par les valeurs contenues dans N . Comme avec la forme canonique, nous employons b_i , c_j et a_{ij} pour noter les termes constants et les coefficients. Nous utilisons en outre v pour désigner un terme constant facultatif dans la fonction objectif. Nous pouvons donc définir, de manière concise, une forme standard par un tuple (N, B, A, b, c, v) , représentant la forme standard

$$z = v + \sum_{j \in N} c_j x_j \quad (29.42)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{pour } i \in B, \quad (29.43)$$

dans laquelle toutes les variables x sont positives. Comme nous soustrayons la somme $\sum_{j \in N} a_{ij} x_j$ dans (29.43), les valeurs a_{ij} sont, en fait, les opposés des coefficients tels qu'ils « apparaissent » dans la forme standard.

Par exemple, dans la forme standard

$$\begin{aligned} z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}, \end{aligned}$$

on a $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = (c_3 \ c_5 \ c_6)^T = (-1/6 \ -1/6 \ -2/3)^T$ et $v = 28$. Notez que les indices dans A , b et c ne sont pas forcément des ensembles d'entiers contigus ; ils dépendent des ensembles d'indices B et N . Comme exemple des valeurs de A qui sont les opposés des coefficients tels que donnés dans la forme standard, observez que l'équation pour x_1 inclut le terme $x_3/6$, alors que le coefficient a_{13} est en fait $-1/6$ et non $+1/6$.

Exercices

29.1.1 Si l'on exprime le programme linéaire (29.24)–(29.28) avec la notation compacte de (29.19)–(29.21), quels sont n , m , A , b et c ?

29.1.2 Donner trois solutions réalisables pour le programme linéaire (29.24)–(29.28). Quelle est la valeur de l'objectif de chacune ?

29.1.3 Pour la forme standard donnée par (29.38)–(29.41), quels sont N , B , A , b , c et v ?

29.1.4 Convertir en forme canonique le programme linéaire suivant :

$$\begin{array}{ll} \text{minimiser} & 2x_1 + 7x_2 \\ \text{en tenant compte des contraintes} & \\ & x_1 = 7 \\ & 3x_1 + x_2 \geq 24 \\ & x_2 \geq 0 \\ & x_3 \leq 0. \end{array}$$

29.1.5 Convertir en forme standard le programme linéaire suivant :

$$\begin{array}{ll} \text{maximiser} & 2x_1 - 6x_3 \\ \text{sous les contraintes} & \\ & x_1 + x_2 - x_3 \leq 7 \\ & 3x_1 - x_2 \geq 8 \\ & -x_1 + 2x_2 + 2x_3 \geq 0 \\ & x_1, x_2, x_3 \geq 0. \end{array}$$

Quelles sont les variables de base et les variables hors-base ?

29.1.6 Montrer que le programme linéaire suivant est irréalisable :

$$\begin{array}{ll} \text{maximiser} & 3x_1 - 2x_2 \\ \text{sous les contraintes} & \\ & x_1 + x_2 \leq 2 \\ & -2x_1 - 2x_2 \leq -10 \\ & x_1, x_2 \geq 0 . \end{array}$$

29.1.7 Montrer que le programme linéaire suivant est non borné :

$$\begin{array}{ll} \text{maximiser} & x_1 - x_2 \\ \text{sous les contraintes} & \\ & -2x_1 + x_2 \leq -1 \\ & -x_1 - 2x_2 \leq -2 \\ & x_1, x_2 \geq 0 . \end{array}$$

29.1.8 Supposez que l'on ait un programme linéaire général à n variables et m contraintes, que l'on convertisse ensuite en forme canonique. Donner une borne supérieure pour le nombre de variables et de contraintes du programme linéaire résultant.

29.1.9 Donner un exemple de programme linéaire dont la région de réalisabilité n'est pas bornée, mais dont la valeur de l'objectif optimale est néanmoins finie.

29.2 FORMULATION DE PROBLÈMES COMME PROGRAMMES LINÉAIRES

Dans ce chapitre nous nous concentrerons sur l'algorithme du simplexe, mais il est important aussi de savoir reconnaître quand un problème peut être formulé en tant que programme linéaire. Une fois un problème exprimé sous la forme d'un programme linéaire de taille polynomiale, on peut le résoudre en temps polynomial à l'aide de l'algorithme de l'ellipsoïde ou du point intérieur. Il existe plusieurs logiciels de programmation linéaire qui permettent de résoudre efficacement les problèmes ; de sorte que, quand le problème a été exprimé sous forme de programme linéaire, ce genre de logiciel permet de le résoudre en pratique.

Nous allons regarder plusieurs exemples concrets de programmation linéaire. Nous commencerons par deux problèmes que nous avons déjà étudiés : le problème des plus courts chemins à origine unique (voir chapitre 24) et le problème du flot maximum (voir chapitre 26). Nous présenterons ensuite le problème du flot maximum à coût minimal. Il existe un algorithme à temps polynomial qui ne s'appuie pas sur la programmation linéaire pour le problème du flot maximum à coût minimal, mais nous

ne l'étudierons pas. Enfin, nous décrirons le problème du flot multi-produits pour lequel le seul algorithme à temps polynomial connu s'appuie sur la programmation linéaire.

a) Plus courts chemins

Le problème des plus courts chemins à origine unique, présenté au chapitre 24, peut se formuler en tant que programme linéaire. Dans cette section, nous nous concentrons sur la formulation du problème du plus court chemin à couple unique, laissant à l'exercice 29.2.3 le soin de traiter le problème plus général des plus courts chemins à origine unique.

Dans le problème du plus court chemin à couple unique, on a un graphe orienté pondéré $G = (V, E)$, avec une fonction de pondération $w : E \rightarrow \mathbf{R}$ qui fait correspondre à chaque arc un poids (à valeur réelle), un sommet origine s et un sommet destination t . On veut calculer la valeur $d[t]$, laquelle est la longueur d'un plus court chemin menant de s à t . Pour pouvoir exprimer ce problème en tant que programme linéaire, il faut déterminer un ensemble de variables et de contraintes qui indiquent quand on a un plus court chemin entre s et t . Heureusement, c'est précisément ce à quoi sert l'algorithme de Bellman-Ford. Quand cet algorithme se termine, il a calculé, pour chaque sommet v , une valeur $d[v]$ telle que, pour tout arc $(u, v) \in E$, on a $d[v] \leq d[u] + w(u, v)$. Le sommet origine reçoit initialement la valeur $d[s] = 0$, qui ne change jamais. On obtient donc le programme linéaire que voici pour calculer la longueur du plus court chemin entre s et t :

$$\text{minimiser } d[t] \quad (29.44)$$

sous les contraintes

$$d[v] \leq d[u] + w(u, v) \quad \text{pour tout arc } (u, v) \in E, \quad (29.45)$$

$$d[s] = 0. \quad (29.46)$$

Dans ce programme linéaire, il y a $|V|$ variables $d[v]$, une pour chaque sommet $v \in V$. Il y a $|E| + 1$ contraintes, une pour chaque arc, plus la contrainte supplémentaire selon laquelle le sommet origine a toujours la valeur 0.

b) Flot maximum

Le problème du flot maximum peut, lui aussi, être exprimé sous la forme d'un programme linéaire. Rappelez-vous que l'on a ici un graphe orienté $G = (V, E)$, dans lequel chaque arc $(u, v) \in E$ possède une capacité positive $c(u, v) \geq 0$, et deux sommets particuliers, à savoir une source s et un puits t . Comme défini à la section 26.1, un flot est une fonction à valeurs réelles $f : V \times V \rightarrow \mathbf{R}$ qui satisfait aux trois propriétés suivantes : contraintes de capacité, symétrie et conservation du flot. Un flot maximum est un flot qui satisfait à ces contraintes et maximise la valeur du flot, laquelle est le flot total provenant de la source. Un flot, par conséquent, satisfait à des contraintes linéaires et la valeur d'un flot est une fonction linéaire. Si l'on se rappelle également que l'on suppose que $c(u, v) = 0$ si $(u, v) \notin E$, on peut exprimer

le problème du flot maximum sous la forme d'un programme linéaire :

$$\text{maximiser } \sum_{v \in V} f(s, v) \quad (29.47)$$

sous les contraintes

$$f(u, v) \leq c(u, v) \quad \text{pour tout } u, v \in V, \quad (29.48)$$

$$f(u, v) = -f(v, u) \quad \text{pour tout } u, v \in V, \quad (29.49)$$

$$\sum_{v \in V} f(u, v) = 0 \quad \text{pour tout } u \in V - \{s, t\}. \quad (29.50)$$

Ce programme linéaire a $|V|^2$ variables correspondant au flot entre chaque couple de sommets, plus $2|V|^2 + |V| - 2$ contraintes.

Il est généralement plus efficace de résoudre un programme linéaire de taille plus petite. À des fins de simplification de la notation, le programme linéaire (29.47)–(29.50) a un flot et une capacité de 0 pour tout couple de sommets u, v avec $(u, v) \notin E$. Il serait plus efficace de réécrire le programme de façon qu'il ait $O(V + E)$ contraintes. C'est ce que vous demandera de faire l'exercice 29.2.5.

c) Flot maximum à coût minimal

Dans cette section, nous avons employé la programmation linéaire pour résoudre des problèmes pour lesquels nous connaissions déjà des algorithmes efficaces. En fait, un algorithme efficace spécifiquement conçu pour un problème, tel l'algorithme de Dijkstra pour les plus courts chemins à origine unique ou la méthode préflot pour le flot maximum, sont souvent plus efficaces que la programmation linéaire, tant en théorie qu'en pratique.

La vraie puissance de la programmation linéaire vient de sa capacité à résoudre de nouveaux problèmes. Rappelez-vous le problème auquel était confronté le candidat à la députation (voir début du chapitre). Pour ce qui est d'obtenir un nombre suffisant de votes sans pour autant dépenser plus d'argent qu'il n'en faut, aucun des algorithmes étudiés jusqu'ici dans ce livre ne résout ce problème ; la programmation linéaire, elle, apporte une solution. Il existe une abondante littérature sur ce genre de problèmes concrets que la programmation linéaire permet de résoudre. La programmation linéaire est aussi des plus utiles pour résoudre des variantes de problèmes pour lesquels on ne connaît peut-être pas encore d'algorithme efficace.

Considérez, par exemple, la généralisation suivante du problème du flot maximal. Supposez que chaque arc (u, v) ait, en plus de sa capacité $c(u, v)$, un coût (à valeur réelle) $a(u, v)$. Si nous envoyons $f(u, v)$ unités de flot sur l'arc (u, v) , nous obtenons un coût de $a(u, v)f(u, v)$. On nous donne aussi une destination de flot d . Nous souhaitons envoyer d unités de flot de s à t d'une façon telle que le coût total induit par le flot, $\sum_{(u,v) \in E} a(u, v)f(u, v)$, soit minimisé. Ce problème porte le nom de **problème du flot à coût minimal**.

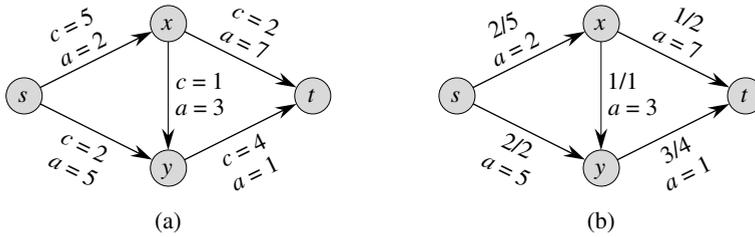


Figure 29.3 (a) Un exemple de problème du flot à coût minimal. On représente les capacités par c et les coûts par a . Le sommet s étant la source et le sommet t étant le puits, on veut envoyer 4 unités de flot de s vers t . (b) Une solution du problème de flot à coût minimal dans lequel on envoie 4 unités de flot de s vers t . Pour chaque arc, le flot et la capacité sont notés sous la forme flot/capacité.

La figure 29.3(a) montre un exemple de problème du flot à coût minimal. On veut envoyer 4 unités de flot de s à t , avec un coût total minimal. Un quelconque flot admissible, c'est-à-dire une fonction f satisfaisant aux contraintes (29.48)–(29.50), induit un coût total $\sum_{(u,v) \in E} a(u, v)f(u, v)$. On veut trouver le flot de 4 unités particulier qui minimise ce coût. Une solution optimale est donnée sur la figure 29.3(b), et son coût total est $\sum_{(u,v) \in E} a(u, v)f(u, v) = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

Il existe des algorithmes à temps polynomial spécifiquement conçus pour le problème du flot à coût minimal, mais ils sortent du cadre de cet ouvrage. On peut, cependant, exprimer le problème du flot à coût minimal sous forme de programme linéaire. Le programme ressemble à celui du problème du flot maximum, avec la contrainte supplémentaire que la valeur du flot doit être exactement de d unités, et avec la nouvelle fonction objectif de minimiser le coût :

$$\text{minimiser} \quad \sum_{(u,v) \in E} a(u, v)f(u, v) \tag{29.51}$$

sous les contraintes

$$f(u, v) \leq c(u, v) \quad \text{pour tout } u, v \in V, \tag{29.52}$$

$$f(u, v) = -f(v, u) \quad \text{pour tout } u, v \in V, \tag{29.53}$$

$$\sum_{v \in V} f(u, v) = 0 \quad \text{pour tout } u \in V - \{s, t\}, \tag{29.54}$$

$$\sum_{v \in V} f(s, v) = d. \tag{29.55}$$

d) Flot multi-produits

Comme dernier exemple, nous allons considérer un autre problème de flot. Supposez que l'entreprise Max & Fils de la section 26.1 décide de diversifier sa gamme de produits et de livrer non plus seulement des tissus, mais aussi des pots et des fleurs artificielles. Chaque type d'équipement est fabriqué dans une usine spécifique, est stocké dans un entrepôt spécifique et doit être expédié chaque jour de l'usine vers

l'entrepôt. Les pots sont fabriqués à Vierzon et envoyés à Mazamet, alors que les fleurs sont fabriquées à Aubagne et expédiées à Privas. La capacité du réseau de transport est toujours la même, cependant, et les différents types de produit, doivent se partager le réseau.

Nous avons ici un exemple de **problème de flot multi-produits**. Dans ce problème, on a encore un graphe orienté $G = (V, E)$ dont chaque arc $(u, v) \in E$ a une capacité positive $c(u, v) \geq 0$. Comme dans le problème du flot maximum, on suppose implicitement que $c(u, v) = 0$ pour $(u, v) \notin E$. En outre, on a k produits différents, K_1, K_2, \dots, K_k , le produit i étant spécifiée *via* le triplet $K_i = (s_i, t_i, d_i)$. s_i est la source du produit i , t_i en est le puits et d_i est la demande, c'est-à-dire la valeur de flot que l'on désire pour le produit i entre s_i et t_i . On définit un flot pour le produit i , noté f_i , (de sorte que $f_i(u, v)$ est le flot du produit i entre le sommet u et le sommet v) comme étant une fonction à valeurs réelles qui satisfait à la conservation de flot, à la symétrie et aux contraintes de capacité. On définit ensuite $f(u, v)$, le **flot total**, comme étant la somme des flots de produit individuels, de sorte que $f(u, v) = \sum_{i=1}^k f_i(u, v)$. Le flot total le long de l'arc (u, v) ne doit pas dépasser la capacité de l'arc (u, v) . Cette contrainte intègre les contraintes de capacité concernant les divers types de produit individuels. De la façon dont le problème est décrit, il n'y a rien à minimiser ; il s'agit seulement de déterminer s'il est possible de trouver un tel flot. Par conséquent, on écrit un programme linéaire avec une fonction objectif « nulle » :

minimiser 0

sous les contraintes

$$\begin{aligned} \sum_{i=1}^k f_i(u, v) &\leq c(u, v) && \text{pour tout } u, v \in V, \\ f_i(u, v) &= -f_i(v, u) && \text{pour tout } i = 1, 2, \dots, k \text{ et} \\ &&& \text{pour tout } u, v \in V, \\ \sum_{v \in V} f_i(u, v) &= 0 && \text{pour tout } i = 1, 2, \dots, k \text{ et} \\ &&& \text{pour tout } u \in V - \{s_i, t_i\}, \\ \sum_{v \in V} f_i(s_i, v) &= d_i && \text{pour tout } i = 1, 2, \dots, k. \end{aligned}$$

Le seul algorithme à temps polynomial connu pour ce problème consiste à l'exprimer en tant que programme linéaire, puis à le résoudre via un algorithme de programmation linéaire à temps polynomial.

Exercices

29.2.1 Mettre sous forme canonique le programme linéaire du plus court chemin à couple unique tel que donné par (29.44)–(29.46).

29.2.2 Écrire explicitement le programme linéaire qui correspond à la recherche du plus court chemin du nœud s vers le nœud y sur la figure 24.2(a).

29.2.3 Dans le problème des plus courts chemins à couple unique, on veut trouver les longueurs des plus courts chemins entre un sommet origine s et tous les sommets $v \in V$. Étant donné un graphe G , écrire un programme linéaire pour lequel la solution a la propriété que $d[v]$ est la longueur du plus court chemin entre s et v pour tout sommet $v \in V$.

29.2.4 Écrire explicitement le programme linéaire qui correspond à la recherche du flot maximal sur la figure 26.1(a).

29.2.5 Réécrire le programme linéaire pour le flot maximum(29.47)–(29.50), de façon qu'il n'utilise que $O(V + E)$ contraintes.

29.2.6 Écrire un programme linéaire qui, étant donné un graphe biparti $G = (V, E)$, résolve le problème du couplage maximum dans un graphe biparti.

29.2.7 Dans le *problème du flot multi-produits à coût minimal*, on a un graphe orienté $G = (V, E)$ dans lequel chaque arc $(u, v) \in E$ possède une capacité positive $c(u, v) \geq 0$ et un coût $a(u, v)$. Comme dans le problème du flot multi-produits, on a k produits différentes, K_1, K_2, \dots, K_k , le produit i étant spécifiée *via* le triplet $K_i = (s_i, t_i, d_i)$. On définit le flot f_i du produit i et le flot total $f(u, v)$ sur l'arc (u, v) comme dans le problème du flot multi-produits. Un flot réalisable est un flot dans lequel le flot total sur chaque arc (u, v) ne dépasse pas la capacité de l'arc (u, v) . Le coût d'un flot est $\sum_{u,v \in V} a(u, v)f(u, v)$, et l'objectif ici est de trouver le flot réalisable de coût minimal. Exprimer ce problème sous forme de programme linéaire.

29.3 ALGORITHME DU SIMPLEXE

L'algorithme du simplexe est la méthode classique de résolution des programmes linéaires. Comparé à la plupart des autres algorithmes donnés dans ce livre, dans le cas le plus défavorable son temps d'exécution n'est pas polynomial. Néanmoins, il permet de se faire une bonne idée de la programmation linéaire et, en pratique, il donne souvent des performances remarquables.

Outre le fait qu'il admet une interprétation géométrique (voir début du chapitre), l'algorithme du simplexe offre une certaine similitude avec l'élimination de Gauss, étudiée à la section 28.3. L'élimination de Gauss commence par un système d'égalités linéaires dont la solution est inconnue. À chaque itération, on réécrit ce système sous une forme équivalente qui a une structure un peu plus développée. Au bout d'un certain nombre d'itérations, on a réécrit le système de telle sorte que la solution est simple à obtenir. L'algorithme du simplexe fonctionne d'une façon similaire, et l'on peut le considérer comme une élimination de Gauss applicable à des inégalités.

Nous allons maintenant décrire l'idée générale qui se cache derrière une itération de l'algorithme du simplexe. À chaque itération est associée une « solution de base », obtenue facilement depuis la forme standard du programme linéaire : on initialise chaque variable hors-base à 0, puis on calcule les valeurs des variables de base à partir des contraintes d'égalité. Une solution de base correspond toujours à un sommet du simplexe. Algébriquement parlant, une itération convertit une forme standard en une forme standard équivalente. La valeur de l'objectif de la solution réalisable de base associée n'est pas inférieure à celle de la précédente itération (généralement, elle est supérieure). Pour parvenir à cet accroissement de la valeur de l'objectif, on choisit une variable hors-base telle que, si l'on devait accroître la valeur de cette variable fixée initialement à 0, alors la valeur de l'objectif augmenterait également. Le montant dont on peut augmenter la variable est limité par les autres contraintes. En particulier, on l'augmente jusqu'à ce qu'une certaine variable de base devienne 0. On réécrit alors la forme standard, échangeant les rôles de cette variable de base et de la variable hors-base sélectionnée. Bien que l'on ait employé une configuration particulière des variables pour guider l'algorithme, et nous utiliserons ce fait dans nos démonstrations, l'algorithme ne conserve pas explicitement cette solution. Il se contente de réécrire le programme linéaire jusqu'à ce que la solution optimale devienne « évidente ».

a) Un exemple d'exécution de l'algorithme du simplexe

Commençons par un exemple détaillé. Soit le programme linéaire que voici, sous forme canonique :

$$\text{maximiser} \quad 3x_1 + x_2 + 2x_3 \quad (29.56)$$

sous les contraintes

$$x_1 + x_2 + 3x_3 \leq 30 \quad (29.57)$$

$$2x_1 + 2x_2 + 5x_3 \leq 24 \quad (29.58)$$

$$4x_1 + x_2 + 2x_3 \leq 36 \quad (29.59)$$

$$x_1, x_2, x_3 \geq 0 \quad (29.60)$$

Pour pouvoir utiliser l'algorithme du simplexe, nous devons mettre le programme linéaire sous forme standard ; nous avons vu comment faire cela à la section 29.1. En plus d'être une manipulation algébrique, la mise sous forme standard est un concept algorithmique utile. En nous rappelant que la section 29.1 a montré que chaque variable a une contrainte de positivité correspondante, nous dirons qu'une contrainte d'égalité est *stricte* pour une configuration particulière de ses variables hors-base si celles-ci forcent la variable de base de la contrainte à devenir 0. De même, toute configuration des variables hors-base qui rendrait négative une variable de base *viole* cette contrainte. Ainsi, les variables d'écart indiquent explicitement le degré de latitude de chaque contrainte et donc aident à savoir de combien on peut accroître les valeurs des variables hors-base sans violer aucune contrainte que ce soit.

En associant les variables d'écart x_4 , x_5 et x_6 aux inégalités (29.57)–(29.59), respectivement, et en mettant le programme linéaire sous forme standard, nous obtenons

$$z = 3x_1 + x_2 + 2x_3 \quad (29.61)$$

$$x_4 = 30 - x_1 - x_2 - 3x_3 \quad (29.62)$$

$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \quad (29.63)$$

$$x_6 = 36 - 4x_1 - x_2 - 2x_3 . \quad (29.64)$$

Le système de contraintes (29.62)–(29.64) a 3 équations et 6 variables. Chaque configuration des variables x_1 , x_2 et x_3 définit des valeurs pour x_4 , x_5 et x_6 ; il y a donc un nombre infini de solutions à ce système d'équations. Une solution est réalisable si x_1, x_2, \dots, x_6 sont tous positifs, et il peut donc y avoir aussi un nombre infini de solutions réalisables. Le fait qu'il y a un nombre infini de solutions possibles à un système de ce genre nous servira dans des démonstrations ultérieures. Nous allons nous concentrer sur la **solution de base** : on met toutes les variables de droite (hors-base) à 0, puis on calcule les valeurs des variables de gauche (de base). Dans cet exemple, la solution de base est $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$, et elle a la valeur de l'objectif $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$. Notez que cette solution de base fait $\bar{x}_i = b_i$ pour tout $i \in B$. Une itération de l'algorithme du simplexe réécrit l'ensemble d'équations et la fonction objectif de façon à mettre un ensemble différent de variables à droite. Chaque réécriture du problème implique donc une solution de base différente. Insistons sur le fait que la réécriture ne modifie en rien le problème de programmation linéaire sous-jacent ; à chaque itération, le problème a le même jeu de solutions réalisables qu'avait le problème de l'itération précédente. En revanche, le problème a une autre solution de base que celle de l'itération précédente.

Si une solution de base est en même temps réalisable, on dit que c'est une **solution réalisable de base**. Pendant l'exécution de l'algorithme du simplexe, la solution de base est presque toujours une solution réalisable de base. La section 29.5 montrera, toutefois, que, pour les quelques premières itérations de l'algorithme du simplexe, la solution de base peut ne pas être réalisable.

Notre but, à chaque itération, consiste à reformuler le programme linéaire de façon à augmenter la valeur de l'objectif de la solution. Nous sélectionnons une variable hors-base x_e dont le coefficient dans la fonction objectif est positif, puis nous augmentons la valeur de x_e autant que faire se peut sans violer aucune des contraintes. La variable x_e devient une variable de base, alors qu'une autre variable x_l devient une variable hors-base. Les valeurs d'autres variables de base et de la fonction objectif peuvent changer elles aussi.

Pour continuer l'exemple, voyons comment nous pouvons augmenter la valeur de x_1 . Quand on augmente x_1 , les valeurs de x_4 , x_5 et x_6 diminuent toutes. Comme nous avons une contrainte de positivité pour chaque variable, aucune d'entre elles ne peut devenir négative. Si x_1 dépasse 30, x_4 devient négative ; pour ce qui concerne x_5 et x_6 ils deviennent négatifs quand x_1 dépasse 12 et 9 respectivement. La troisième contrainte (29.64) est la plus stricte, et c'est donc elle qui définit le degré de latitude

dont nous disposons pour augmenter x_1 . Nous permuterons donc les rôles de x_1 et x_6 . Nous résolvons l'équation (29.64) pour x_1 et obtenons

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}. \quad (29.65)$$

Pour réécrire les autres équations avec x_6 à droite, nous remplaçons x_1 selon le modèle de l'équation (29.65). En faisant cette substitution dans l'équation (29.62), nous obtenons

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}. \end{aligned} \quad (29.66)$$

De la même façon, nous pouvons combiner l'équation (29.65) avec la contrainte (29.63) et la fonction objectif (29.61) pour réécrire notre programme linéaire sous la forme suivante :

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \quad (29.67)$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \quad (29.68)$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \quad (29.69)$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}. \quad (29.70)$$

Cette opération porte le nom de *pivot*. Comme on vient de le voir, un pivot choisit une variable hors-base x_e , dite *variable entrante*, et une variable de base x_l , dite *variable sortante*, dont il permute les rôles.

Le programme linéaire décrit par (29.67)–(29.70) équivaut au programme linéaire décrit par (29.61)–(29.64). Les opérations effectuées dans l'algorithme du simplexe sont des réécritures d'équation ayant pour but de faire passer des variables entre les côtés gauche et droit et de remplacer une équation par une autre. La première opération crée de façon triviale un problème équivalent, et la seconde, *via* de l'algèbre linéaire élémentaire, crée aussi un problème équivalent.

Pour montrer cette équivalence, notez que notre solution de base originelle (0, 0, 0, 30, 24, 36) satisfait aux nouvelles équations (29.68)–(29.70) avec la valeur de l'objectif $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$. La solution de base associée au nouveau programme linéaire met les valeurs hors-base à 0 et vaut (9, 0, 0, 21, 6, 0), avec la valeur de l'objectif $z = 27$. Un calcul arithmétique simple prouve que cette solution satisfait également aux équations (29.62)–(29.64) et que, quand on la combine avec la fonction objectif (29.61), elle possède la valeur de l'objectif $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$.

En continuant l'exemple, nous voulons trouver une nouvelle variable dont nous pourrions augmenter la valeur. Nous ne voulons pas accroître x_6 car, si sa valeur augmente, la valeur de l'objectif diminue. Nous pouvons essayer d'augmenter x_2 ou x_3 ;

nous prendrons x_3 . Dans quelle mesure pouvons-nous accroître x_3 sans violer quelque contrainte que ce soit ? La contrainte (29.68) impose la limite 18, la contrainte (29.69) impose la limite $42/5$ et la contrainte (29.70) impose la limite $3/2$. Ici aussi, c'est la troisième contrainte qui est la plus stricte ; nous allons donc réécrire la troisième contrainte de telle sorte que x_3 passe sur le côté gauche et x_5 sur le côté droit. Nous substituerons ensuite la nouvelle équation dans les équations (29.67)–(29.69) pour obtenir le nouveau système équivalent

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \quad (29.71)$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \quad (29.72)$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \quad (29.73)$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} . \quad (29.74)$$

Ce système a la solution de base associée $(33/4, 0, 3/2, 69/4, 0, 0)$, avec la valeur de l'objectif $111/4$. Maintenant la seule façon d'accroître la valeur de l'objectif, c'est d'augmenter x_2 . Les trois contraintes donnent des limites supérieures valant respectivement 132, 4 et ∞ . (La limite supérieure ∞ dans la contrainte (29.74) vient de ce que, quand on augmente x_2 , la valeur de la variable de base x_4 augmente aussi. Cette contrainte n'impose donc aucune restriction pour l'accroissement de x_2 .) Nous augmentons x_2 pour lui donner la valeur 4, et cette variable devient hors-base. Ensuite, nous résolvons l'équation (29.73) pour x_2 puis, par remplacement dans les autres équations, obtenons

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \quad (29.75)$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \quad (29.76)$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \quad (29.77)$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} . \quad (29.78)$$

À ce stade, tous les coefficients de la fonction objectif sont négatifs. Comme nous le verrons plus loin dans ce chapitre, cette situation ne se produit que quand on a réécrit le programme linéaire de telle sorte que la solution de base soit une solution optimale. Ainsi, pour ce problème, la solution $(8, 4, 0, 18, 0, 0)$, donnant la valeur de l'objectif 28, est optimale. Nous pouvons maintenant revenir à notre programme linéaire d'origine, donné par (29.56)–(29.60). Comme les seules variables du programme original sont x_1, x_2 et x_3 , notre solution est $x_1 = 8, x_2 = 4$ et $x_3 = 0$, avec la valeur de l'objectif $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Notez que les valeurs des variables d'écart dans la solution finale mesurent le degré de latitude de chaque inégalité. La variable d'écart x_4 vaut 18 ; dans l'inégalité (29.57), le membre gauche, qui vaut $8 + 4 + 0 = 12$, est inférieur

de 18 au membre droit qui vaut 30. Les variables d'écart x_5 et x_6 valent 0 et, en vérité, dans les inégalités (29.58) et (29.59), les membres gauche et droit sont égaux. Observez également que, même si dans la forme relâchée originelle les coefficients sont entiers, ce n'est pas forcément le cas des coefficients des autres programmes linéaires, et que les solutions intermédiaires ne sont pas forcément entières. Qui plus est, la solution finale d'un programme linéaire n'est pas obligée d'être entière ; c'est pure coïncidence si, dans cet exemple, la solution est à valeurs entières.

b) Pivote

Nous allons maintenant formaliser le mécanisme du pivot. La procédure PIVOT prend en entrée une forme standard, donnée par le n-uplet (N, B, A, b, c, v) , l'indice l de la variable sortante x_l et l'indice e de la variable entrante x_e . Elle retourne le n-uplet $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$ décrivant la nouvelle forme standard. (Rappelons d'ailleurs que les éléments des matrices A et \hat{A} sont en fait les opposés des coefficients figurant dans la forme standard.)

PIVOT(N, B, A, b, c, v, l, e)

- 1 ▷ Calcule les coefficients de l'équation pour nouvelle variable de base x_e .
- 2 $\hat{b}_e \leftarrow b_l / a_{le}$
- 3 **pour** tout $j \in N - \{e\}$
- 4 **faire** $\hat{a}_{ej} \leftarrow a_{lj} / a_{le}$
- 5 $\hat{a}_{el} \leftarrow 1 / a_{le}$
- 6 ▷ Calcule les coefficients des contraintes restantes.
- 7 **pour** tout $i \in B - \{l\}$
- 8 **faire** $\hat{b}_i \leftarrow b_i - a_{ie} \hat{b}_e$
- 9 **pour** tout $j \in N - \{e\}$
- 10 **faire** $\hat{a}_{ij} \leftarrow a_{ij} - a_{ie} \hat{a}_{ej}$
- 11 $\hat{a}_{il} \leftarrow -a_{ie} \hat{a}_{el}$
- 12 ▷ Calcule la fonction objectif.
- 13 $\hat{v} \leftarrow v + c_e \hat{b}_e$
- 14 **pour** tout $j \in N - \{e\}$
- 15 **faire** $\hat{c}_j \leftarrow c_j - c_e \hat{a}_{ej}$
- 16 $\hat{c}_l \leftarrow -c_e \hat{a}_{el}$
- 17 ▷ Calcule nouveaux ensembles de variables de base/hors-base.
- 18 $\hat{N} = N - \{e\} \cup \{l\}$
- 19 $\hat{B} = B - \{l\} \cup \{e\}$
- 20 **retourner** $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$

PIVOT fonctionne de la manière suivante. Les lignes 2–5 calculent les coefficients de x_e dans la nouvelle équation en réécrivant l'équation qui a x_l dans son membre de gauche de façon qu'elle ait, à la place, x_e dans son membre de gauche. Les lignes 7–11 actualisent les autres équations en remplaçant chaque occurrence de x_e par le membre

droit de cette nouvelle équation. Les lignes 13–16 font la même substitution pour la fonction objectif, alors que les lignes 18 et 19 actualisent l'ensemble des variables hors-base et celui des variables de base. La ligne 20 donne la nouvelle forme standard. Sous cette forme, si $a_{le} = 0$, PIVOT provoque une erreur de division par 0 ; mais, comme nous le verrons dans les démonstrations des lemmes 29.2 et 29.12, PIVOT n'est appelée que quand $a_{le} \neq 0$.

Résumons maintenant l'effet de PIVOT sur les valeurs des variables dans la solution de base.

Lemme 29.1 *Considérons un appel à PIVOT(N, B, A, b, c, v, l, e) dans lequel $a_{le} \neq 0$. Soient $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$ les valeurs retournées par l'appel, et soit \bar{x} la solution de base après l'appel. Alors*

- 1) $\bar{x}_j = 0$ pour tout $j \in \widehat{N}$.
- 2) $\bar{x}_e = b_l/a_{le}$.
- 3) $\bar{x}_i = b_i - a_{ie}\widehat{b}_e$ pour tout $i \in \widehat{B} - \{e\}$.

Démonstration : La première assertion découle de ce que la solution de base met toujours toutes les variables hors-base à 0. Quand nous réglons chaque variable hors-base sur 0 dans une contrainte

$$\bar{x}_i = \widehat{b}_i - \sum_{j \in N} \widehat{a}_{ij} \bar{x}_j,$$

nous avons $\bar{x}_i = \widehat{b}_i$ pour tout $i \in \widehat{B}$. Comme $e \in \widehat{B}$, d'après la ligne 2 de PIVOT, on a

$$\bar{x}_e = \widehat{b}_e = b_l/a_{le},$$

ce qui prouve la deuxième assertion. De même, en utilisant la ligne 8 pour tout $i \in \widehat{B} - e$, on a

$$\bar{x}_i = \widehat{b}_i = b_i - a_{ie}\widehat{b}_e,$$

ce qui prouve la troisième assertion. □

c) Formalisation de l'algorithme du simplexe

Nous voici prêt à formaliser l'algorithme du simplexe, que nous avons prouvé par l'exemple. Cet exemple était particulièrement sympathique, et nous aurions pu avoir à résoudre plusieurs autres problèmes :

- Comment déterminer si un programme linéaire est réalisable ?
- Que faire si le programme linéaire est réalisable, mais que la solution de base initiale n'est pas réalisable ?
- Comment déterminer si un programme linéaire est non borné ?
- Comment choisir les variables entrantes et sortantes ?

À la section 29.5, nous montrerons comment déterminer si un problème est réalisable, et si oui, comment trouver une forme standard dans laquelle la solution de base initiale est réalisable. Nous supposons donc ici que nous avons une procédure INITIALISE-SIMPLEXE(A, b, c) qui prend en entrée un programme linéaire exprimé sous forme canonique, c'est-à-dire une matrice $m \times n$ nommée par exemple $A = (a_{ij})$, un vecteur à m dimensions $b = (b_i)$ et un vecteur à n dimensions $c = (c_j)$. Si le problème est irréalisable, la procédure retourne un message indiquant que le programme n'est pas réalisable puis se termine. Autrement, elle produit une forme standard dont la solution de base initiale est réalisable.

La procédure SIMPLEXE prend en entrée un programme linéaire sous forme canonique, comme précédemment indiqué. Elle retourne un n -vecteur $\bar{x} = (\bar{x}_j)$ qui est une solution optimale pour le programme linéaire défini par (29.19)–(29.21).

SIMPLEXE(A, b, c)

```

1  ( $N, B, A, b, c, v$ ) ← INITIALISE-SIMPLEXE( $A, b, c$ )
2  tant que qu'un indice  $j \in N$  vérifie  $c_j > 0$ 
3      faire choisir un indice  $e \in N$  pour lequel  $c_e > 0$ 
4          pour tout indice  $i \in B$ 
5              faire si  $a_{ie} > 0$ 
6                  alors  $\Delta_i \leftarrow b_i/a_{ie}$ 
7                  sinon  $\Delta_i \leftarrow \infty$ 
8              choisir un indice  $l \in B$  qui minimise  $\Delta_i$ 
9              si  $\Delta_l = \infty$ 
10                 alors retourner « non borné »
11                 sinon ( $N, B, A, b, c, v$ ) ← PIVOT( $N, B, A, b, c, v, l, e$ )
12 pour  $i \leftarrow 1$  à  $n$ 
13     faire si  $i \in B$ 
14         alors  $\bar{x}_i \leftarrow b_i$ 
15         sinon  $\bar{x}_i \leftarrow 0$ 
16 retourner ( $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ )

```

Voici comment fonctionne la procédure SIMPLEXE. En ligne 1, elle appelle la procédure INITIALISE-SIMPLEXE(A, b, c), précédemment mentionnée qui, soit détermine que le programme linéaire est irréalisable, soit retourne une forme standard pour laquelle la solution de base est réalisable. L'essentiel de l'algorithme est représenté par la boucle **tant que** des lignes 2–11. Si dans la fonction objectif tous les coefficients sont négatifs, alors la boucle **tant que** se termine. Autrement, en ligne 3, on sélectionne une variable x_e dont le coefficient dans la fonction objective est positif pour qu'elle soit la variable entrante. Bien que nous puissions choisir n'importe quelle variable conforme comme variable entrante, nous supposons que l'on utilise une règle déterministe spécifiée à l'avance. Ensuite, sur les lignes 4–8, nous contrôlons chaque contrainte et prenons celle qui limite le plus la quantité dont nous pouvons accroître x_e sans violer l'une quelconque des contraintes de positivité; la

variable de base associée à cette contrainte est x_l . Ici aussi, on pourrait avoir la liberté de choisir entre plusieurs variables pour prendre la variable sortante, mais nous supposons que l'on utilise une règle déterministe spécifiée par avance. Si aucune des contraintes ne limite la quantité dont la variable entrante peut augmenter, l'algorithme retourne « non borné » en ligne 10. Sinon, la ligne 11 permute les rôles des variables entrante et sortante en appelant la sous-routine PIVOT(N, B, A, b, c, v, l, e), précédemment décrite. Les lignes 12–15 calculent une solution pour les variables originelles du programme linéaire $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ en mettant toutes les variables hors-base à 0 et en donnant à \bar{x}_i la valeur b_i pour chaque variable de base. Dans le théorème 29.10, nous verrons que cette solution est une solution optimale pour le programme linéaire. Enfin, la ligne 16 retourne les valeurs calculées de ces variables originelles de programmation linéaire.

Pour prouver que SIMPLEXE est correcte, nous commencerons par montrer que, si SIMPLEXE a une solution réalisable initiale et qu'il se termine effectivement, alors soit il retourne une solution réalisable, soit il détermine que le programme linéaire est non borné. Ensuite, nous montrerons que SIMPLEXE se termine effectivement. Enfin, à la section 29.4, nous prouverons que la solution obtenue est optimale.

Lemme 29.2 *Étant donné un programme linéaire (A, b, c) , supposez que l'appel à INITIALISE-SIMPLEXE sur la ligne 1 de SIMPLEXE retourne une forme standard pour laquelle la solution de base est réalisable. Alors, si SIMPLEXE retourne une solution en ligne 16, cette solution est une solution réalisable pour le programme linéaire. Si SIMPLEXE retourne « non borné » en ligne 10, alors le programme linéaire est non borné.*

Démonstration :

Nous utiliserons l'invariant de boucle en trois parties suivant : Au début de chaque itération de la boucle **tant que** des lignes 2–11,

- 1) la forme standard équivaut à la forme standard que produit l'appel à INITIALISE-SIMPLEXE,
- 2) pour tout $i \in B$, on a $b_i \geq 0$, et
- 3) la solution de base associée à la forme standard est réalisable.

Initialisation : L'équivalence de la forme standard est chose triviale pour la première itération. On suppose, dans l'énoncé du lemme, que l'appel à INITIALISE-SIMPLEXE sur la ligne 1 de SIMPLEXE retourne une forme standard pour laquelle la solution de base est réalisable. Ainsi, la troisième partie de l'invariant est vraie. En outre, comme chaque variable de base x_i est initialisée à b_i dans la solution de base, et que la réalisabilité de la solution de base implique que chaque variable de base x_i est positive, on a $b_i \geq 0$. Ainsi, la deuxième partie de l'invariant est vérifiée.

Conservation : Nous allons maintenant montrer que l'invariant de boucle est conservé quand il n'y a pas exécution de l'instruction **retourner** de la ligne 10.

Nous traiterons le cas de l'exécution de la ligne 10 quand nous étudierons l'arrêt de la procédure.

Une itération de la boucle **tant que** permute les rôles d'une variable de base et d'une variable hors-base. Les seules opérations effectuées sont des résolutions d'équation et la substitution d'une équation dans une autre ; par conséquent, la forme standard équivaut à celle de la précédente itération qui, de par l'invariant de boucle, équivaut à la forme standard initiale.

Nous allons maintenant démontrer la deuxième partie de l'invariant de boucle. On suppose que, au début de chaque itération de la boucle **tant que**, $b_i \geq 0$ pour tout $i \in B$, et l'on doit montrer que ces inégalités restent vraies après l'appel à PIVOT en ligne 11. Comme les seules modifications des variables b_i et de l'ensemble B des variables de base se produisent dans cette affectation, il suffit de montrer que la ligne 11 conserve cette partie de l'invariant. Soient b_i, a_{ij} et B les valeurs avant l'appel de PIVOT, et soient \hat{b}_i les valeurs retournées par PIVOT.

Remarquons, pour commencer, que $\hat{b}_e \geq 0$ car $b_l \geq 0$ d'après l'invariant de boucle, $a_{le} > 0$ d'après la ligne 5 de SIMPLEXE, et $\hat{b}_e = b_l/a_{le}$ d'après la ligne 2 de PIVOT.

Pour les autres indices $i \in B - l$, nous avons

$$\begin{aligned}\hat{b}_i &= b_i - a_{ie}\hat{b}_e && \text{(d'après la ligne 8 de PIVOT)} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(d'après la ligne 2 de PIVOT)} .\end{aligned}\quad (29.79)$$

Nous avons deux cas à considérer, selon que $a_{ie} > 0$ ou $a_{ie} \leq 0$. Si $a_{ie} > 0$, alors, comme on a choisi l de telle façon que

$$b_l/a_{le} \leq b_i/a_{ie} \quad \text{pour tout } i \in B , \quad (29.80)$$

on a

$$\begin{aligned}\hat{b}_i &= b_i - a_{ie}(b_l/a_{le}) && \text{(d'après l'équation (29.79))} \\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(d'après l'inégalité (29.80))} \\ &= b_i - b_i \\ &= 0 ,\end{aligned}$$

ce qui entraîne que $\hat{b}_i \geq 0$. Si $a_{ie} \leq 0$, alors, comme a_{le}, b_i et b_l sont tous positifs, l'équation (29.79) implique que \hat{b}_i doit être, lui aussi, positif.

Nous allons maintenant montrer que la solution de base est réalisable, c'est-à-dire que toutes les variables ont des valeurs positives. Les variables hors-base sont mises à 0 et, partant, sont positives. Chaque variable de base x_i est définie par l'équation

$$x_i = b_i - \sum_{j \in N} a_{ij}x_j .$$

La solution de base fait $\bar{x}_i = b_i$. En utilisant la deuxième partie de l'invariant de boucle, nous arrivons à la conclusion que chaque variable de base \bar{x}_i est positive.

Terminaison : La boucle **tant que** peut se terminer de deux façons. Si elle se termine à cause de la condition de la ligne 2, alors la solution de base courante est réalisable et la procédure retourne cette solution via la ligne 16. L'autre façon de se terminer est de retourner « non borné » sur la ligne 10. Dans ce cas, pour chaque

itération de la boucle **pour** des lignes 4–7, quand la ligne 5 est exécutée, on trouve que $a_{ie} \leq 0$. Soit x la solution de base associée à la forme standard du début de l'itération qui a retourné « non borné ». Considérons la solution \bar{x} définie ainsi

$$\bar{x}_i = \begin{cases} \infty & \text{si } i = e, \\ 0 & \text{si } i \in N - \{e\}, \\ b_i - \sum_{j \in N} a_{ij} \bar{x}_j & \text{si } i \in B. \end{cases}$$

Nous allons maintenant montrer que cette solution est réalisable, c'est à dire que toutes les variables sont positives. Les variables hors-base autres que \bar{x}_e sont 0, alors que \bar{x}_e est positive ; par conséquent, toutes les variables hors-base sont positives. Pour chaque variable de base \bar{x}_i , on a

$$\begin{aligned} \bar{x}_i &= b_i - \sum_{j \in N} a_{ij} \bar{x}_j \\ &= b_i - a_{ie} \bar{x}_e. \end{aligned}$$

L'invariant de boucle implique $b_i \geq 0$, et l'on a $a_{ie} \leq 0$ et $\bar{x}_e = \infty > 0$. Par conséquent, $\bar{x}_i \geq 0$.

Maintenant, nous allons montrer que la valeur de l'objectif pour la solution \bar{x} est non bornée. La valeur de l'objectif est

$$\begin{aligned} z &= v + \sum_{j \in N} c_j \bar{x}_j \\ &= v + c_e \bar{x}_e. \end{aligned}$$

Puisque $c_e > 0$ (d'après la ligne 3) et que $\bar{x}_e = \infty$, la valeur de l'objectif est ∞ , ce qui entraîne que le programme linéaire est non borné. □

À chaque itération, SIMPLEXE conserve A , b , c et v en plus des ensembles N et B . La conservation explicite de A , b , c et v est essentielle pour l'efficacité de l'algorithme du simplexe, mais elle n'est pas absolument indispensable. En d'autres termes, la forme standard est déterminée de manière unique par l'ensemble des variables de base et par l'ensemble des variables hors-base. Avant de prouver cela, nous allons démontrer un lemme algébrique utile.

Lemme 29.3 *Soit I un ensemble d'indices. Pour tout $i \in I$, soient α_i et β_i des réels, et soit x_i une variable à valeur réelle. Soit γ un réel quelconque. Supposons que, pour toute configuration des x_i , l'on ait*

$$\sum_{i \in I} \alpha_i x_i = \gamma + \sum_{i \in I} \beta_i x_i. \tag{29.81}$$

Alors $\alpha_i = \beta_i$ pour tout $i \in I$, et $\gamma = 0$.

Démonstration : Comme l'équation (29.81) est vraie pour des valeurs quelconques des x_i , on peut utiliser des valeurs particulières afin d'en tirer des conclusions concernant α , β et γ . Si nous faisons $x_i = 0$ pour tout $i \in I$, nous concluons que $\gamma = 0$. Prenons maintenant un indice arbitraire $i \in I$, puis faisons $x_i = 1$ et $x_k = 0$ pour tout $k \neq i$. Alors, nous devons avoir $\alpha_i = \beta_i$, comme nous avons pris i quelconque dans I , nous en concluons que $\alpha_i = \beta_i$ pour tout $i \in I$. □

Montrons maintenant que la forme standard d'un programme linéaire est déterminée de manière unique par l'ensemble des variables de base.

Lemme 29.4 *Soit (A, b, c) un programme linéaire sous forme canonique. Étant donné un ensemble B de variables de base, il y a unicité de la forme standard associée.*

Démonstration : Raisonnons par l'absurde en supposant qu'il existe deux formes standard différentes avec le même ensemble B de variables de base. Les formes standard doivent aussi avoir des ensembles identiques $N = \{1, 2, \dots, n+m\} - B$ de variables hors-base. Nous écrivons la première forme standard comme suit

$$z = v + \sum_{j \in N} c_j x_j \quad (29.82)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{pour } i \in B, \quad (29.83)$$

et la seconde comme suit

$$z = v' + \sum_{j \in N} c'_j x_j \quad (29.84)$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{pour } i \in B. \quad (29.85)$$

Considérons le système d'équations formé en soustrayant chaque équation de la ligne (29.85) de l'équation homologue de la ligne (29.83). Le système résultant est

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \quad \text{pour } i \in B$$

ou, de manière équivalente,

$$\sum_{j \in N} a_{ij} x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij} x_j \quad \text{pour } i \in B.$$

Maintenant, pour tout $i \in B$, appliquons le lemme 29.3 avec $\alpha_i = a_{ij}$, $\beta_i = a'_{ij}$ et $\gamma = b_i - b'_i$. Comme $\alpha_i = \beta_i$, nous avons $a_{ij} = a'_{ij}$ pour tout $j \in N$; et comme $\gamma = 0$, nous avons $b_i = b'_i$. Par conséquent, pour les deux formes standard, A et b sont identiques à A' et b' . Via un raisonnement similaire, l'exercice 29.3.1 montrera que l'on a aussi $c = c'$ et $v = v'$, et donc que les formes standard sont forcément identiques. \square

Reste à prouver que SIMPLEXE se termine effectivement et que la solution alors obtenue est optimale. La section 29.4 traitera de l'optimalité. Pour l'instant, occupons-nous de vérifier que la procédure se termine.

d) Terminaison de l'algorithme

Dans l'exemple donné au début de cette section, chaque itération de l'algorithme du simplexe augmentait la valeur de l'objectif associée à la solution de base. L'exercice 29.3.2 vous demandera de prouver qu'aucune itération de SIMPLEXE ne peut diminuer la valeur de l'objectif associée à la solution de base. En revanche, il se peut

qu'une itération laisse la valeur de l'objectif inchangée. Ce phénomène porte le nom de *dégénérescence*, et nous allons maintenant l'étudier de manière très détaillée.

La valeur de l'objectif est modifiée suite à l'affectation $\widehat{v} \leftarrow v + c_e \widehat{b}_e$ en ligne 13 de PIVOT. Comme SIMPLEXE appelle PIVOT uniquement quand $c_e > 0$, le seul cas où la valeur de l'objectif ne bouge pas (c'est-à-dire, quand $\widehat{v} = v$) est celui où \widehat{b}_e est 0. Cette valeur est assignée via $\widehat{b}_e \leftarrow b_l/a_{le}$ en ligne 2 de PIVOT. Comme on appelle toujours PIVOT avec $a_{le} \neq 0$, on voit que, pour que \widehat{b}_e soit égal à 0 et donc pour que la valeur de l'objectif reste inchangée, il faut que $b_l = 0$.

Cette situation peut se produire. Considérez le programme linéaire

$$\begin{aligned} z &= && x_1 &+& x_2 &+& x_3 \\ x_4 &= 8 &-& x_1 &-& x_2 && \\ x_5 &= && x_2 &-& x_3 &.& \end{aligned}$$

Supposez que nous choissions x_1 comme variable entrante et x_4 comme variable sortante. Après opération de pivot, nous obtenons

$$\begin{aligned} z &= 8 && &+& x_3 &-& x_4 \\ x_1 &= 8 &-& x_2 && &-& x_4 \\ x_5 &= && x_2 &-& x_3 &.& \end{aligned}$$

À ce stade, notre seul choix est de pivoter avec x_3 entrante et x_5 sortante. Comme $b_5 = 0$, la valeur de l'objectif qui est 8 reste inchangée après l'opération de pivot :

$$\begin{aligned} z &= 8 &+& x_2 &-& x_4 &-& x_5 \\ x_1 &= 8 &-& x_2 &-& x_4 && \\ x_3 &= && x_2 && &-& x_5 &.& \end{aligned}$$

La valeur de l'objectif n'a pas changé, mais notre représentation a changé. Heureusement, si nous refaisons un pivot avec x_2 en entrée et x_1 en sortie, la valeur de l'objectif augmente et l'algorithme du simplexe peut se poursuivre.

Montrons maintenant que la dégénérescence est la seule chose qui pourrait empêcher l'algorithme du simplexe de se terminer. Rappelez-vous que nous avons supposé que SIMPLEXE choisissait les indices e et l , sur les lignes 3 et 8 respectivement, en se basant sur une certaine règle déterministe. Nous dirons que SIMPLEXE *boucle* si les formes standard de deux itérations différentes sont identiques, auquel cas, comme SIMPLEXE est un algorithme déterministe, il boucle ad vitam aeternam à travers la même suite de formes standard.

Lemme 29.5 Si SIMPLEXE n'arrive pas à se terminer en au plus $\binom{n+m}{m}$ itérations, alors il boucle.

Démonstration : D'après le lemme 29.4, l'ensemble B des variables de base détermine avec unicité une forme standard. Il y a $n + m$ variables et $|B| = m$, et donc il y a $\binom{n+m}{m}$ façons de choisir B . Par conséquent, il n'y a que $\binom{n+m}{m}$ formes standard différentes. Donc, si SIMPLEXE effectue plus de $\binom{n+m}{m}$ itérations, c'est qu'il boucle. \square

Bien que le bouclage soit théoriquement possible, c'est un phénomène extrêmement rare. On peut l'éviter en choisissant les variables entrante et sortante avec un peu plus de soin. Une option consiste à perturber légèrement l'entrée de façon qu'il soit impossible d'avoir deux solutions ayant la même valeur de l'objectif. Une deuxième possibilité consiste à choisir la variable entrante de manière lexicographique, et une troisième option consiste à choisir la variable entrante en choisissant toujours la variable de plus petit indice. Cette dernière stratégie porte le nom de **règle de Bland** . Nous nous passerons de démontrer que ces stratégies préviennent tout bouclage.

Lemme 29.6 *Si sur les lignes 3 et 8 de SIMPLEXE, les arbitrages se font toujours via sélection de la variable de plus petit indice, alors SIMPLEXE se termine.*

Nous terminerons cette section par le lemme suivant.

Lemme 29.7 *Si INITIALISE-SIMPLEXE retourne une forme standard pour laquelle la solution de base est réalisable, alors soit SIMPLEXE signale qu'un programme linéaire est non borné, soit il se termine avec une solution réalisable en au plus $\binom{n+m}{m}$ itérations.*

Démonstration : Les lemmes 29.2 et 29.6 montrent que, si INITIALISE-SIMPLEXE retourne une forme standard pour laquelle la solution de base est réalisable, soit SIMPLEXE signale qu'un programme linéaire est non borné, soit il se termine sur une solution réalisable. La contraposée du lemme 29.5 montre que, si SIMPLEXE se termine sur une solution réalisable, alors il se termine en au plus $\binom{n+m}{m}$ itérations. \square

Exercices

29.3.1 Compléter la démonstration du lemme 29.4 en montrant que l'on a forcément aussi $c = c'$ et $v = v'$.

29.3.2 Montrer que l'appel à PIVOT sur la ligne 11 de SIMPLEXE ne diminue jamais la valeur de v .

29.3.3 On suppose que l'on convertit un programme linéaire (A, b, c) de la forme canonique vers la forme standard. Montrer que la solution de base est réalisable si et seulement si $b_i \geq 0$ pour $i = 1, 2, \dots, m$.

29.3.4 Résoudre le programme linéaire suivant en utilisant SIMPLEXE :

maximiser $18x_1 + 12.5x_2$

sous les contraintes

$$\begin{aligned} x_1 + x_2 &\leq 20 \\ x_1 &\leq 12 \\ x_2 &\leq 16 \\ x_1, x_2 &\geq 0 \end{aligned}$$

29.3.5 Résoudre le programme linéaire suivant en utilisant SIMPLEXE :

maximiser $-5x_1 - 3x_2$

sous les contraintes

$$\begin{aligned} x_1 - x_2 &\leq 1 \\ 2x_1 + x_2 &\leq 2 \\ x_1, x_2 &\geq 0 \end{aligned}$$

29.3.6 Résoudre le programme linéaire suivant en utilisant SIMPLEXE :

minimiser $x_1 + x_2 + x_3$

sous les contraintes

$$\begin{aligned} 2x_1 + 7.5x_2 + 3x_3 &\geq 10000 \\ 20x_1 + 5x_2 + 10x_3 &\geq 30000 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

29.4 DUALITÉ

Nous avons prouvé que, moyennant certaines hypothèses, SIMPLEXE se termine effectivement. Cependant, nous n'avons pas encore montré qu'elle trouve bien la solution optimale d'un programme linéaire. Pour ce faire, nous allons introduire un concept puissant baptisé *dualité en programmation linéaire*.

La dualité est une propriété très importante. Dans un problème d'optimisation, l'identification d'un problème dual va presque toujours de pair avec la découverte d'un algorithme à temps polynomial. La puissance de la dualité vient aussi de sa capacité à fournir la preuve qu'une solution est vraiment optimale.

Supposez par exemple que, étant donné un problème de flot maximum, on trouve un flot f de valeur $|f|$. Comment savoir si f est un flot maximum ? D'après le théorème de flot maximum-coupe minimum (théorème 26.7), si l'on peut trouver une coupe dont la valeur est aussi $|f|$, alors on a vérifié que f est bien un flot maximal. Voilà un exemple de dualité : à un problème de maximisation donné, on associe un problème de minimisation tel que les deux problèmes aient les mêmes valeurs optimales de l'objectif.

Partant d'un programme linéaire de maximisation, il faut formuler un programme linéaire *dual* dans lequel l'objectif est de minimiser et dont la valeur optimale est égale à celle du programme linéaire originel. Quand on parle de programmes linéaires duaux, on dit que le programme originel est le programme linéaire *primal*.

Étant donné un programme linéaire primal sous forme canonique, comme dans (29.16)–(29.18), on définit le programme linéaire dual de la manière suivante

$$\text{minimiser} \quad \sum_{i=1}^m b_i y_i \quad (29.86)$$

sous les contraintes

$$\sum_{i=1}^m a_{ij} y_i \geq c_j \quad \text{pour } j = 1, 2, \dots, n, \quad (29.87)$$

$$y_i \geq 0 \quad \text{pour } i = 1, 2, \dots, m. \quad (29.88)$$

Pour former le dual, on remplace la maximisation par une minimisation, on permute les rôles des membres de droite et des coefficients de la fonction objectif et l'on remplace l'inégalité inférieur ou égal par une inégalité supérieur ou égal. Chacune des m contraintes du primal a une variable associée y_i dans le dual, et chacune des n contraintes du dual a une variable associée x_j dans le primal. Par exemple, considérez le programme linéaire (29.56)–(29.60). Son dual est

$$\text{minimiser} \quad 30y_1 + 24y_2 + 36y_3 \quad (29.89)$$

sous les contraintes

$$y_1 + 2y_2 + 4y_3 \geq 3 \quad (29.90)$$

$$y_1 + 2y_2 + y_3 \geq 1 \quad (29.91)$$

$$3y_1 + 5y_2 + 2y_3 \geq 2 \quad (29.92)$$

$$y_1, y_2, y_3 \geq 0. \quad (29.93)$$

Nous montrerons, avec le théorème 29.10, que la valeur optimale du programme dual est toujours égale à la valeur optimale du programme primal. En outre, l'algorithme du simplexe résout en fait implicitement tant le programme linéaire primal que le programme dual simultanément, fournissant ce faisant une preuve de l'optimalité.

Commençons par démontrer la *dualité faible*, qui s'exprime ainsi : toute solution réalisable du programme linéaire primal a une valeur qui est inférieure à celle d'une quelconque solution réalisable du programme dual.

Lemme 29.8 (Dualité faible en programmation linéaire) *Soit \bar{x} une quelconque solution réalisable du programme linéaire primal (29.16)–(29.18) et soit \bar{y} une quelconque solution réalisable du programme linéaire dual (29.86)–(29.88). Alors*

$$\sum_{j=1}^n c_j \bar{x}_j \leq \sum_{i=1}^m b_i \bar{y}_i.$$

Démonstration : On a

$$\begin{aligned} \sum_{j=1}^n c_j \bar{x}_j &\leq \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij} \bar{y}_i \right) \bar{x}_j \quad (\text{d'après (29.87)}) \\ &= \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij} \bar{x}_j \right) \bar{y}_i \\ &\leq \sum_{i=1}^m b_i \bar{y}_i \quad (\text{d'après (29.17)}) . \quad \square \end{aligned}$$

Corollaire 29.9 Soit \bar{x} une solution réalisable d'un programme linéaire primal (A, b, c) , et soit \bar{y} une solution réalisable du programme dual associé. Si

$$\sum_{j=1}^n c_j \bar{x}_j = \sum_{i=1}^m b_i \bar{y}_i$$

alors \bar{x} et \bar{y} sont des solutions optimales pour les programmes linéaires primal et dual respectivement.

Démonstration : D'après le lemme 29.8, la valeur de l'objectif d'une solution réalisable du primal est inférieure à celle d'une solution réalisable du dual. Le programme linéaire primal est un problème de maximisation, alors que le dual est un problème de minimisation. Par conséquent, si les solutions réalisables \bar{x} et \bar{y} ont la même valeur de l'objectif, il est impossible d'améliorer l'une ou l'autre. \square

Avant de prouver qu'il existe toujours une solution duale dont la valeur est égale à celle d'une solution primale optimale, expliquons comment trouver une telle solution. Quand nous exécutons l'algorithme du simplexe pour le programme linéaire (29.56)–(29.60), l'itération finale donnait la forme standard (29.75)–(29.78) avec $B = \{1, 2, 4\}$ et $N = \{3, 5, 6\}$. Comme nous le verrons ci-après, la solution de base associée à la forme standard finale est une solution optimale pour le programme linéaire ; une solution optimale pour le programme linéaire (29.56)–(29.60) est donc $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (8, 4, 0)$ avec la valeur de l'objectif $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Comme nous le verrons également ci-après, nous pouvons en déduire une solution duale optimale : les opposés des coefficients de la fonction objectif primale sont les valeurs des variables duales. Plus précisément, supposons que la dernière forme standard du primal soit

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ x_i &= b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{pour } i \in B . \end{aligned}$$

Alors, une solution duale optimale consiste à définir

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{si } (n+i) \in N , \\ 0 & \text{sinon .} \end{cases} \quad (29.94)$$

Par conséquent, une solution optimale du programme dual défini en (29.89)–(29.93) est $\bar{y}_1 = 0$ (comme $n + 1 = 4 \in B$), $\bar{y}_2 = -c'_5 = 1/6$ et $\bar{y}_3 = -c'_6 = 2/3$. En évaluant la fonction objectif duale (29.89), on obtient une valeur de l'objectif de $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$, ce qui confirme que la valeur de l'objectif du primal est bien égale à la valeur de l'objectif du dual. En combinant ces calculs avec le lemme 29.8, on a une preuve que la valeur de l'objectif optimale du programme primal est 28. Montrons maintenant que, en général, cette méthode permet d'obtenir une solution optimale pour le dual, ainsi qu'une preuve de l'optimalité d'une solution du primal.

Théorème 29.10 (Dualité en programmation linéaire) *Supposez que SIMPLEXE retourne les valeurs $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ pour le programme primal (A, b, c) . Soient N et B l'ensemble des variables hors-base et l'ensemble des variables de base pour la forme standard finale, soit c' les coefficients dans la forme standard finale, et soit $\bar{y} = (\bar{y}_1, \bar{y}_2, \dots, \bar{y}_m)$ défini par l'équation (29.94). Alors \bar{x} est une solution optimale pour le programme primal, \bar{y} est une solution optimale pour le dual, et*

$$\sum_{j=1}^n c_j \bar{x}_j = \sum_{i=1}^m b_i \bar{y}_i. \quad (29.95)$$

Démonstration :

D'après le corollaire 29.9, si nous pouvons trouver des solutions réalisables \bar{x} et \bar{y} satisfaisant à l'équation (29.95), alors \bar{x} et \bar{y} sont forcément des solutions primale et duale optimales. Montrons donc que les solutions \bar{x} et \bar{y} décrites dans l'énoncé du théorème satisfont à l'équation (29.95).

Supposez que l'on exécute SIMPLEXE sur un programme primal, tel qu'indiqué sur les lignes (29.16)–(29.18). L'algorithme passe par une suite de formes standard jusqu'à ce qu'il se termine avec une forme standard finale ayant la fonction objectif

$$z = v' + \sum_{j \in N} c'_j x_j. \quad (29.96)$$

Comme SIMPLEXE s'est terminé avec une solution, d'après la condition de la ligne 2 nous savons que

$$c'_j \leq 0 \quad \text{pour tout } j \in N. \quad (29.97)$$

Si nous définissons

$$c'_j = 0 \quad \text{pour tout } j \in B, \quad (29.98)$$

nous pouvons réécrire l'équation (29.96) comme suit

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j \quad (\text{car } c'_j = 0 \text{ si } j \in B) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \quad (\text{car } N \cup B = \{1, 2, \dots, n+m\}). \end{aligned} \quad (29.99)$$

Pour la solution de base \bar{x} associée à cette forme standard finale, $\bar{x}_j = 0$ pour tout $j \in N$ et $z = v'$. Puisque toutes les formes standard sont équivalentes, si l'on évalue la fonction objectif originelle sur \bar{x} , on doit obtenir la même valeur de l'objectif, c'est-à-dire

$$\sum_{j=1}^n c_j \bar{x}_j = v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j \quad (29.100)$$

$$\begin{aligned} &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot 0) + \sum_{j \in B} (0 \cdot \bar{x}_j) \\ &= v' . \end{aligned} \quad (29.101)$$

Montrons maintenant que \bar{y} , défini par l'équation (29.94), est réalisable pour le programme linéaire dual et que sa valeur de l'objectif $\sum_{i=1}^m b_i \bar{y}_i$ est égale à $\sum_{j=1}^n c_j \bar{x}_j$. L'équation (29.100) dit que la première et la dernière formes standard, évaluées en \bar{x} , sont égales. Plus généralement, l'équivalence de toutes les formes standard entraîne que, pour *tout* ensemble de valeurs $x = (x_1, x_2, \dots, x_n)$, on a

$$\sum_{j=1}^n c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j .$$

Par conséquent, pour tout ensemble de valeurs x , on a

$$\begin{aligned} &\sum_{j=1}^n c_j x_j \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{j=n+1}^{n+m} c'_j x_j \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m c'_{n+i} x_{n+i} \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m (-\bar{y}_i) x_{n+i} && \text{(d'après l'équation (29.94))} \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m (-\bar{y}_i) \left(b_i - \sum_{j=1}^n a_{ij} x_j \right) && \text{(d'après l'équation (29.32))} \\ &= v' + \sum_{j=1}^n c'_j x_j - \sum_{i=1}^m b_i \bar{y}_i + \sum_{i=1}^m \sum_{j=1}^n (a_{ij} x_j) \bar{y}_i \\ &= v' + \sum_{j=1}^n c'_j x_j - \sum_{i=1}^m b_i \bar{y}_i + \sum_{j=1}^n \sum_{i=1}^m (a_{ij} \bar{y}_i) x_j \\ &= \left(v' - \sum_{i=1}^m b_i \bar{y}_i \right) + \sum_{j=1}^n \left(c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i \right) x_j , \end{aligned}$$

de sorte que

$$\sum_{j=1}^n c_j x_j = \left(v' - \sum_{i=1}^m b_i \bar{y}_i \right) + \sum_{j=1}^n \left(c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i \right) x_j. \quad (29.102)$$

En appliquant le lemme 29.3 à l'équation (29.102), nous obtenons

$$v' - \sum_{i=1}^m b_i \bar{y}_i = 0, \quad (29.103)$$

$$c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i = c_j \quad \text{pour } j = 1, 2, \dots, n. \quad (29.104)$$

D'après l'équation (29.103), on a $\sum_{i=1}^m b_i \bar{y}_i = v'$; la valeur de l'objectif du dual ($\sum_{i=1}^m b_i \bar{y}_i$) est donc égale à celle du primal (v'). Reste à prouver que la solution \bar{y} est réalisable pour le problème dual. D'après (29.97) et (29.98), on a $c'_j \leq 0$ pour tout $j = 1, 2, \dots, n + m$. Donc, pour $i = 1, 2, \dots, m$, (29.104) implique que

$$\begin{aligned} c_j &= c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i \\ &\leq \sum_{i=1}^m a_{ij} \bar{y}_i, \end{aligned}$$

ce qui satisfait aux contraintes (29.87) du dual. Enfin, comme $c'_j \leq 0$ pour tout $j \in N \cup B$, quand on initialise \bar{y} selon l'équation (29.94), on a chaque $\bar{y}_i \geq 0$; par conséquent, les contraintes de positivité sont satisfaites elles aussi. \square

Nous avons démontré que, étant donné un programme linéaire réalisable, si INITIALISE-SIMPLEXE retourne une solution réalisable et si SIMPLEXE se termine sans retourner « non borné », alors la solution obtenue est bien une solution optimale. Nous avons également montré comment construire une solution optimale pour le programme linéaire dual.

Exercices

29.4.1 Formuler le dual du programme linéaire de l'exercice 29.3.4.

29.4.2 On suppose que l'on a un programme linéaire qui n'est pas sous forme canonique. On pourrait produire le dual en commençant par convertir en forme canonique, puis en prenant le dual. Il serait cependant plus commode de produire directement le dual. Expliquer comment, à partir d'un programme linéaire donné, on peut produire directement le dual associé.

29.4.3 Écrire le dual du programme linéaire de flot maximum défini en (29.47)–(29.50). Expliquer comment on peut interpréter cette formulation en tant que problème de coupe minimum.

29.4.4 Écrire le dual du programme linéaire du flot de coût minimum défini en (29.51)–(29.55). Expliquer comment on peut interpréter ce problème en termes de graphes et de flots.

29.4.5 Montrer que le dual du dual d'un programme linéaire est le programme primal.

29.4.6 Quel résultat du chapitre 26 peut-on interpréter comme étant de la dualité faible pour le problème du flot maximum ?

29.5 SOLUTION DE BASE RÉALISABLE INITIALE

Dans cette section, nous allons d'abord montrer comment vérifier qu'un programme linéaire est réalisable et, si tel est le cas, comment produire une forme standard pour laquelle la solution de base est réalisable. Nous terminerons en démontrant le théorème fondamental de la programmation linéaire, qui énonce que la procédure SIMPLEXE donne toujours un résultat correct.

a) Recherche d'une solution initiale

À la section 29.3, nous sommes partis du principe que nous avons une procédure INITIALISE-SIMPLEXE qui détermine si un programme linéaire a des solutions réalisables et, si tel est le cas, qui donne une forme standard pour laquelle la solution de base est réalisable. Nous allons maintenant décrire cette procédure.

Ce n'est pas parce qu'un programme linéaire est réalisable que la solution de base initiale l'est forcément. Considérez, par exemple, le programme linéaire suivant :

$$\text{maximiser} \quad 2x_1 - x_2 \quad (29.105)$$

sous les contraintes

$$2x_1 - x_2 \leq 2 \quad (29.106)$$

$$x_1 - 5x_2 \leq -4 \quad (29.107)$$

$$x_1, x_2 \geq 0 \quad (29.108)$$

Si l'on devait convertir ce programme linéaire en forme standard, la solution de base ferait les assignations $x_1 = 0$ et $x_2 = 0$. Comme cette solution viole la contrainte (29.107), ce n'est donc pas une solution réalisable. Par conséquent, INITIALISE-SIMPLEXE ne peut même pas retourner la forme standard évidente. En y regardant de plus près, on se sait pas trop si ce programme linéaire a même des solutions réalisables. Pour savoir s'il en a effectivement, on peut formuler un **programme linéaire auxiliaire**. Pour ce programme linéaire auxiliaire, on saura trouver (moyennant un peu de travail) une forme standard pour laquelle la solution de base est réalisable. En outre, la solution de ce programme linéaire auxiliaire déterminera si le programme linéaire initial est réalisable et, si tel est le cas, fournira une solution réalisable permettant d'initialiser SIMPLEXE.

Lemme 29.11 Soit L un programme linéaire sous forme canonique, donné comme dans (29.16)–(29.18). Soit L_{aux} le programme linéaire suivant à $n + 1$ variables :

$$\text{maximiser} \quad -x_0 \quad (29.109)$$

sous les contraintes

$$\sum_{j=1}^n a_{ij}x_j - x_0 \leq b_i \quad \text{pour } i = 1, 2, \dots, m, \quad (29.110)$$

$$x_j \geq 0 \quad \text{pour } j = 0, 1, \dots, n. \quad (29.111)$$

Alors, L est réalisable si et seulement si la valeur de l'objectif optimale de L_{aux} est 0.

Démonstration : Supposons que L ait une solution réalisable $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$. Alors, la solution $\bar{x}_0 = 0$ combinée avec \bar{x} est une solution réalisable de L_{aux} avec la valeur de l'objectif 0. Comme $x_0 \geq 0$ est une contrainte de L_{aux} et que la fonction objectif consiste à maximiser $-x_0$, cette solution est forcément optimale pour L_{aux} .

Inversement, supposons que la valeur de l'objectif optimale de L_{aux} soit 0. Alors, $\bar{x}_0 = 0$ et les valeurs des autres variables \bar{x} satisfont aux contraintes de L . \square

Nous allons maintenant exposer notre stratégie de recherche d'une solution réalisable de base initiale pour un programme linéaire L sous forme canonique :

INITIALISE-SIMPLEXE(A, b, c)

- 1 soit l l'indice du b_i minimal
- 2 **si** $b_l \geq 0$ ▷ Est-ce que la solution de base initiale est réalisable ?
- 3 **alors retourner** ($\{1, 2, \dots, n\}, \{n + 1, n + 2, \dots, n + m\}, A, b, c, 0$)
- 4 former L_{aux} en ajoutant $-x_0$ au membre gauche de chaque équation et en choisissant la fonction objectif sur $-x_0$
- 5 soit (N, B, A, b, c, v) la forme standard résultante pour L_{aux}
- 6 ▷ L_{aux} a $n + 1$ variables hors-base et m variables de base.
- 7 $(N, B, A, b, c, v) \leftarrow \text{PIVOT}(N, B, A, b, c, v, l, 0)$
- 8 ▷ La solution de base est maintenant réalisable pour L_{aux} .
- 9 itérer la boucle **tant que** des lignes 2–11 de SIMPLEXE jusqu'à obtention d'une solution optimale pour L_{aux}
- 10 **si** la solution de base donne $\bar{x}_0 = 0$
- 11 **alors retourner** la forme standard finale en supprimant x_0 et en restaurant la fonction objectif d'origine
- 12 **sinon retourner** « irréalisable »

INITIALISE-SIMPLEXE fonctionne de la manière suivante. Sur les lignes 1–3, on teste implicitement la solution de base de la forme standard initiale de L donnée par $N = \{1, 2, \dots, n\}$, $B = \{n + 1, n + 2, \dots, n + m\}$, $\bar{x}_i = b_i$ pour tout $i \in B$ et $\bar{x}_j = 0$ pour tout $j \in N$. (Créer la forme standard ne demande aucun travail explicite, vu que les valeurs de A , b et c sont les mêmes pour la forme standard et pour la forme canonique.) Si cette solution de base est réalisable, c'est-à-dire si $\bar{x}_i \geq 0$ pour

tout $i \in N \cup B$, alors la procédure retourne cette forme standard. Autrement, sur la ligne 4, on construit le programme linéaire auxiliaire L_{aux} comme indiqué dans le lemme 29.11. Puisque la solution de base initiale de L n'est pas réalisable, la solution de base initiale de la forme standard de L_{aux} n'est pas non plus réalisable. Sur la ligne 7, on effectue donc un appel à PIVOT, avec x_0 entrante et x_l sortante, l'indice l étant choisi en ligne 1 comme étant l'indice du b_i le plus négatif. Nous verrons bientôt que la solution de base résultant de cet appel à PIVOT est réalisable. Maintenant que l'on a une forme standard pour laquelle la solution de base est réalisable, on peut, en ligne 9, appeler derechef PIVOT jusqu'à résolution complète du programme linéaire auxiliaire. Le test de la ligne 10 repose sur le principe suivant : si l'on trouve une solution optimale pour L_{aux} ayant la valeur de l'objectif 0, alors en ligne 11 on crée une forme standard de L pour laquelle la solution de base est réalisable. Pour ce faire, on supprime tous les termes x_0 dans les contraintes et l'on rétablit la fonction objectif initiale de L . La fonction objectif d'origine peut contenir à la fois des variables de base et des variables hors-base. Par conséquent, dans la fonction objectif on remplace chaque variable de base par le membre droit de sa contrainte associée. Si, en revanche, la ligne 10 montre que le programme linéaire originel L est irréalisable, alors on retourne cette information sur la ligne 12.

Nous allons maintenant montrer le fonctionnement de INITIALISE-SIMPLEXE sur l'exemple du programme linéaire (29.105)–(29.108). Ce programme linéaire sera réalisable si nous trouvons des valeurs positives pour x_1 et x_2 qui satisfassent aux inégalités (29.106) et (29.107). En utilisant le lemme 29.11, nous formulons le programme linéaire auxiliaire

$$\text{maximiser} \quad -x_0 \quad (29.112)$$

sous les contraintes

$$2x_1 - x_2 - x_0 \leq 2 \quad (29.113)$$

$$x_1 - 5x_2 - x_0 \leq -4 \quad (29.114)$$

$$x_1, x_2, x_0 \geq 0 .$$

D'après le lemme 29.11, si la valeur de l'objectif optimale de ce programme auxiliaire est 0, alors le programme originel a une solution réalisable. Si la valeur de l'objectif optimale de ce programme auxiliaire est négative, alors le programme linéaire originel n'a pas de solution réalisable.

En écrivant ce programme linéaire sous forme standard, nous obtenons

$$\begin{aligned} z &= && -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0 . \end{aligned}$$

Nous ne sommes pas encore tirés d'affaire puisque la solution de base, qui ferait l'assignation $x_4 = -4$, n'est pas réalisable pour ce programme linéaire auxiliaire. Nous pouvons, toutefois, via un appel à PIVOT, convertir cette forme standard en une forme dans laquelle la solution de base est réalisable. Comme indiqué en ligne 7, nous

choisissons x_0 comme variable entrante. En ligne 1, nous prenons comme variable sortante x_4 qui est la variable de base dont la valeur dans la solution de base est la plus négative. Après pivotement, nous avons la forme standard

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4 . \end{aligned}$$

La solution de base associée est $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$, qui est réalisable. Nous appelons ensuite PIVOT de manière répétée, jusqu'à obtenir une solution optimale pour L_{aux} . Ici, un seul appel à PIVOT avec x_2 entrante et x_0 sortante donne

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} . \end{aligned}$$

Cette forme standard est la solution finale du problème auxiliaire. Comme cette solution a $x_0 = 0$, nous savons que notre problème initial était réalisable. En outre, comme $x_0 = 0$, on peut sans problème la supprimer de l'ensemble des contraintes. Nous pouvons alors utiliser la fonction objectif d'origine, en opérant des substitutions appropriées de façon à n'inclure que des variables hors-base. Dans notre exemple, nous obtenons la fonction objectif

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \right) .$$

En faisant $x_0 = 0$ et en simplifiant, nous obtenons la fonction objectif

$$\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} ,$$

et la forme standard

$$\begin{aligned} z &= \frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} . \end{aligned}$$

Cette forme standard a une solution de base réalisable, que nous pouvons retourner à la procédure SIMPLEXE.

Nous allons maintenant prouver la validité de INITIALISE-SIMPLEXE.

Lemme 29.12 *Si un programme linéaire L n'a pas de solution réalisable, alors INITIALISE-SIMPLEXE retourne « Irréalisable ». Sinon, la procédure retourne une forme standard valide pour laquelle la solution de base est réalisable.*

Démonstration : Supposons d'abord que le programme linéaire L n'ait pas de solution réalisable. Alors, d'après le lemme 29.11, la valeur de l'objectif optimale de L_{aux} ,

défini dans (29.109)–(29.111), est non nulle, et d'après la contrainte de positivité pour x_0 , une solution optimale doit avoir une valeur de l'objectif strictement négative. En outre, cette valeur de l'objectif doit être finie, puisque faire $x_i = 0$, pour $i = 1, 2, \dots, n$, et $x_0 = |\min_{i=1}^m \{b_i\}|$ donne une solution réalisable, et cette solution a la valeur de l'objectif $-|\min_{i=1}^m \{b_i\}|$. Par conséquent, la ligne 9 de INITIALISE-SIMPLEXE trouve une solution avec une valeur de l'objectif strictement négative. Soit \bar{x} la solution de base associée à la forme standard finale. On ne peut pas avoir $\bar{x}_0 = 0$, car alors L_{aux} aurait la valeur de l'objectif 0, ce qui est contradiction avec le fait que la valeur de l'objectif est strictement négative. Le test de la ligne 10 aboutit donc à faire retourner « irréalisable » en ligne 12.

Supposons maintenant que le programme linéaire L ait bien une solution réalisable. D'après l'exercice 29.3.3, nous savons que si $b_i \geq 0$ pour $i = 1, 2, \dots, m$, alors la solution de base associée à la forme standard initiale est réalisable. Dans ce cas, les lignes 2–3 retournent la forme standard associée à l'entrée. (Il n'y a pas grand chose à faire pour convertir la forme canonique en forme standard, vu que A , b et c sont les mêmes dans les deux cas.)

Dans la suite de la démonstration, nous traiterons le cas où le programme linéaire est réalisable mais où la procédure ne rend pas la main en ligne 3. Nous montrerons qu'en pareil cas les lignes 4–9 trouvent une solution réalisable pour L_{aux} avec la valeur de l'objectif 0. Tout d'abord, d'après les lignes 1–2, on doit avoir

$$b_l < 0,$$

et

$$b_l \leq b_i \quad \text{pour tout } i \in B. \quad (29.115)$$

En ligne 7, on effectue une opération de pivot dans laquelle la variable sortante x_l est le membre gauche de l'équation avec b_l minimum, et la variable entrante est x_0 , qui est la variable supplémentaire. Montrons maintenant que, après ce pivot, toutes les données contenues dans b sont positives, et donc que la solution de base pour L_{aux} est réalisable. Si \bar{x} désigne la solution de base après l'appel à PIVOT, et si \hat{b} et \hat{B} désignent les valeurs retournées par PIVOT, alors le lemme 29.1 implique

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\hat{b}_e & \text{si } i \in \hat{B} - \{e\}, \\ b_l/a_{le} & \text{si } i = e. \end{cases} \quad (29.116)$$

L'appel à PIVOT en ligne 7 a $e = 0$, et d'après (29.110), nous avons

$$a_{i0} = a_{ie} = -1 \quad \text{pour tout } i \in B. \quad (29.117)$$

(Notez que a_{i0} est le coefficient de x_0 tel qu'il apparaît dans (29.110), et non pas l'opposé du coefficient, car L_{aux} est en forme canonique et non en forme standard.) Comme $l \in B$, nous avons également $a_{le} = -1$. Ainsi, $b_l/a_{le} > 0$, et par conséquent $\bar{x}_e > 0$. Pour les autres variables de base, on a

$$\begin{aligned} \bar{x}_i &= b_i - a_{ie}\hat{b}_e && \text{(d'après l'équation (29.116))} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(d'après la ligne 2 de PIVOT)} \\ &= b_i - b_l && \text{(d'après l'équation (29.117) et } a_{le} = -1) \\ &\geq 0 && \text{(d'après l'inégalité (29.115))}, \end{aligned}$$

Cela entraîne que chaque variable de base est désormais positive. De ce fait, la solution de base après l'appel à PIVOT en ligne 7 est réalisable. On exécute ensuite la

ligne 9, qui résout L_{aux} . Comme nous avons supposé que L a une solution réalisable, le lemme 29.11 implique que L_{aux} a une solution optimale avec la valeur de l'objectif 0. Puisque toutes les formes standards sont équivalentes, la solution de base finale de L_{aux} a forcément $\bar{x}_0 = 0$; et après suppression de x_0 dans le programme linéaire, nous obtenons une forme standard qui est réalisable pour L . La procédure retourne alors cette forme standard en ligne 10. \square

b) Théorème fondamental de la programmation linéaire

Nous terminerons ce chapitre en démontrant que la procédure SIMPLEXE fonctionne. En particulier, un programme linéaire quelconque est soit irréalisable, soit non borné, soit pourvu d'une solution optimale ayant une valeur de l'objectif finie; et dans tous les cas, SIMPLEXE fonctionne correctement.

Théorème 29.13 (Théorème fondamental de la programmation linéaire) *Un programme linéaire L quelconque, exprimé sous forme canonique, vérifie l'une des trois conditions suivantes :*

- 1) *il a une solution optimale avec une valeur de l'objectif finie,*
- 2) *il est irréalisable,*
- 3) *il est non borné.*

Si L est irréalisable, SIMPLEXE retourne « irréalisable ». Si L est non borné, SIMPLEXE retourne « non borné ». Autrement, SIMPLEXE retourne une solution optimale ayant une valeur de l'objectif finie.

Démonstration : D'après le lemme 29.12, si le programme linéaire L est irréalisable, alors SIMPLEXE retourne « irréalisable ». Supposons maintenant que le programme linéaire L soit réalisable. D'après le lemme 29.12, INITIALISE-SIMPLEXE retourne une forme standard pour laquelle la solution de base est réalisable. Par conséquent, d'après le lemme 29.7, SIMPLEXE soit retourne « non borné », soit se termine sur une solution réalisable. Si elle se termine avec une solution finie, alors le théorème 29.10 nous dit que cette solution est optimale. Si, en revanche, SIMPLEXE retourne « non borné », le lemme 29.2 nous dit que le programme linéaire L est effectivement non borné. Comme SIMPLEXE se termine toujours de l'une de ces façons, le théorème est démontré. \square

Exercices

29.5.1 Donner un pseudo code détaillé qui implémente les lignes 5 et 11 de INITIALISE-SIMPLEXE.

29.5.2 Montrer que, quand INITIALISE-SIMPLEXE exécute la boucle principale de SIMPLEXE, elle ne retourne jamais « non borné ».

29.5.3 Soit un programme linéaire L donné sous forme canonique tel que, pour L comme pour son dual, les solutions de base associées aux formes standard initiales soient réalisables. Montrer que la valeur de l'objectif optimale de L est alors 0.

29.5.4 Supposez que l'on autorise les inégalités strictes dans les programmes linéaires. Montrer qu'alors le théorème fondamental de la programmation linéaire n'est plus vrai.

29.5.5 Résoudre le programme linéaire suivant à l'aide de SIMPLEXE :

$$\begin{array}{l} \text{maximiser} \quad x_1 + 3x_2 \\ \text{sous les contraintes} \\ \quad -x_1 + x_2 \leq -1 \\ \quad -2x_1 - 2x_2 \leq -6 \\ \quad -x_1 + 4x_2 \leq 2 \\ \quad x_1, x_2 \geq 0. \end{array}$$

29.5.6 Résoudre le programme linéaire donné par (29.6)–(29.10).

29.5.7 Soit le programme linéaire à 1 variable suivant, appelé P :

$$\begin{array}{l} \text{maximiser} \quad tx \\ \text{sous les contraintes} \\ \quad rx \leq s \\ \quad x \geq 0, \end{array}$$

r, s et t étant des réels arbitraires. Soit D le dual de P . Quelles sont les valeurs de r, s et t pour lesquelles vous pouvez affirmer que

- 1) P et D ont des solutions optimales avec des valeurs de l'objectif finies.
- 2) P est réalisable, mais pas D .
- 3) D est réalisable, mais pas P .
- 4) ni P ni D n'est réalisable.

PROBLÈMES

29.1. Réalisabilité d'inégalités linéaires

Étant données m inégalités linéaires à n variables x_1, x_2, \dots, x_n , le **problème de la réalisabilité d'inégalités linéaires** concerne le fait de savoir s'il existe une configuration des variables qui satisfasse simultanément à toutes les inégalités.

- a.** Montrer que, si l'on a un algorithme de programmation linéaire, on peut l'employer pour résoudre le problème de la réalisabilité d'inégalités linéaires. Le nombre de variables et de contraintes utilisées dans le problème de programmation linéaire doit être polynomial en n et m .
- b.** Montrer que, si l'on a un algorithme pour le problème de la réalisabilité d'inégalités linéaires, on peut s'en servir pour résoudre un problème de programmation

linéaire. Le nombre de variables et d'inégalités linéaires utilisées dans le problème de la réalisabilité d'inégalités linéaires doit être polynomial en n et m , lesquels désignent respectivement le nombre de variables et le nombre de contraintes du programme linéaire.

29.2. Écarts complémentaires

Les *écarts complémentaires* décrivent une relation entre les valeurs des variables primales et des contraintes duales et entre les valeurs des variables duales et des contraintes primales. Soit \bar{x} une solution réalisable d'un programme linéaire primal donné par (29.16)–(29.18), et soit \bar{y} une solution réalisable du programme linéaire dual donné par (29.86)–(29.88). Les écarts complémentaires disent que les conditions suivantes sont nécessaires et suffisantes pour que \bar{x} et \bar{y} soient optimales :

$$\sum_{i=1}^m a_{ij}\bar{y}_i = c_j \text{ ou } \bar{x}_j = 0 \text{ pour } j = 1, 2, \dots, n$$

et

$$\sum_{j=1}^n a_{ij}\bar{x}_j = b_i \text{ ou } \bar{y}_i = 0 \text{ pour } i = 1, 2, \dots, m.$$

- a. Vérifier que les écarts complémentaires restent vrais pour le programme linéaire donné par (29.56)–(29.60).
- b. Démontrer que les écarts complémentaires restent vrais pour tout programme linéaire primal et pour le dual associé.
- c. Démontrer qu'une solution réalisable \bar{x} d'un programme linéaire primal donné par les lignes (29.16)–(29.18) est optimale si et seulement si il existe des valeurs $\bar{y} = (\bar{y}_1, \bar{y}_2, \dots, \bar{y}_m)$ telles que
 - 1) \bar{y} est une solution réalisable du programme linéaire dual donné en (29.86)–(29.88),
 - 2) $\sum_{i=1}^m a_{ij}\bar{y}_i = c_j$ chaque fois que $\bar{x}_j > 0$, et
 - 3) $\bar{y}_i = 0$ chaque fois que $\sum_{j=1}^n a_{ij}\bar{x}_j < b_i$.

29.3. Programmation linéaire entière

Un *problème de programmation linéaire entier* est un problème de programmation linéaire ayant comme contrainte supplémentaire le fait que les variables x doivent prendre des valeurs entières. L'exercice 34.5.3 montre que le simple fait de déterminer si un programme linéaire entier a une solution réalisable relève de la NP-complétude, ce qui implique qu'il est peu probable qu'il existe un algorithme à temps polynomial pour ce problème.

- a. Montrer que la dualité faible (lemme 29.8) s'applique à un programme linéaire entier.

- b.** Montrer que la dualité (théorème 29.10) ne s'applique pas toujours à un programme linéaire entier.
- c.** Étant donné un programme linéaire primal sous forme canonique, soient P la valeur de l'objectif optimale pour le primal, D la valeur de l'objectif optimale pour le dual, IP la valeur de l'objectif optimale pour la version entière du primal (à savoir, le primal complété par la contrainte que les variables prennent des valeurs entières) et ID la valeur de l'objectif optimale de la version entière du dual. Montrer que, si le programme entier primal et le programme entier dual sont tous les deux réalisables et bornés, alors

$$IP \leq P = D \leq ID .$$

29.4. Lemme de Farkas

Soient A une matrice $m \times n$ et c un n -vecteur. Alors, le lemme de Farkas dit qu'un et un seul des systèmes

$$\begin{aligned} Ax &\leq 0, \\ c^T x &> 0 \end{aligned}$$

et

$$\begin{aligned} A^T y &= c, \\ y &\geq 0 \end{aligned}$$

est résoluble, x étant un n -vecteur et y un m -vecteur. Démontrer le lemme de Farkas.

NOTES

Ce chapitre n'a fait qu'effleurer le vaste champ de la programmation linéaire. Il existe plusieurs ouvrages consacrés exclusivement à la programmation linéaire, dont ceux de Chvátal [62], Gass [111], Karloff [171], Schrijver [266], et Vanderbei [304]. Il existe beaucoup d'autres livres contenant une bonne étude de la programmation linéaire, dont ceux de Papadimitriou et Steiglitz [237] et Ahuja, Magnanti, et Orlin [7]. La présentation adoptée dans ce chapitre s'inspire de Chvátal.

L'algorithme du simplexe a été découvert par G. Dantzig en 1947. Peu de temps après, on trouva qu'un certain nombre de problèmes appartenant à des domaines variés pouvaient s'exprimer en tant que programme linéaires et être résolus via l'algorithme du simplexe. Cette découverte favorisa grandement l'usage de la programmation linéaire et de plusieurs algorithmes. Les variantes de l'algorithme du simplexe sont toujours les méthodes préférées pour la résolution des problèmes de programmation linéaire. Cet historique est décrit à plusieurs endroits, notamment dans les notes de [62] et [171].

L'algorithme de l'ellipsoïde fut le premier algorithme à temps polynomial de la programmation linéaire, et on le doit à L. G. Khachian en 1979 ; cette découverte s'appuyait sur des travaux antérieurs de N. Z. Shor, D. B. Judin et A. S. Nemirovskii. L'utilisation de cet algorithme pour résoudre toutes sortes de problèmes d'optimisation combinatoire est décrite dans les ouvrages de Grötschel, Lovász et Schrijver [134]. De nos jours, cet algorithme ne semble pas être aussi utilisé en pratique que l'algorithme du simplexe.

L'ouvrage de Karmarkar [172] contient une description de son algorithme de point intérieur. Par la suite, bien d'autres chercheurs ont inventé des algorithmes à point intérieur. Vous trouverez de bonnes études sur ce thème dans l'article de Goldfarb et Todd [122], ainsi que dans le livre de Ye [319].

L'analyse de l'algorithme du simplexe est un domaine de recherche foisonnant. Klee et Minty ont construit un exemple pour lequel l'algorithme s'exécute en $2^n - 1$ itérations. L'algorithme du simplexe donne généralement de très bonnes performances en pratique et beaucoup de chercheurs ont essayé de donner une justification théorique à ces observations empiriques. Une ligne de recherche, initiée par Borgwardt et continuée par beaucoup d'autres, montre que, moyennant certaines hypothèses probabilistes concernant l'entrée, l'algorithme du simplexe converge en temps polynomial moyen. De récents progrès dans ce domaine ont été réalisés par Spielman et Teng [284], qui introduisent « l'analyse lissée des algorithmes » et l'appliquent à l'algorithme du simplexe.

L'algorithme du simplexe est connu pour s'exécuter plus efficacement dans certains cas particuliers. Citons, tout particulièrement, l'algorithme de flot du simplexe, qui est l'algorithme du simplexe appliqué aux problèmes de flots dans les réseaux. Pour certains problèmes de flot, dont le problème des plus courts chemins, le problème du flot maximal et le problème du flot à coût minimal, il existe des variantes de l'algorithme de flot du simplexe qui s'exécutent en temps polynomial. Reportez-vous, par exemple, à l'article de Orlin [234] et aux références qu'il renferme.

Chapitre 30

Polynômes et transformée rapide de Fourier

La méthode directe pour additionner deux polynômes de degré n requiert un temps $\Theta(n)$, tandis que la méthode directe pour multiplier deux polynômes de degré n prend un temps $\Theta(n^2)$. Dans ce chapitre, nous allons montrer comment la transformée rapide de Fourier, ou FFT (Fast Fourier Transform), peut ramener le temps de multiplication de deux polynômes à $\Theta(n \lg n)$.

Les transformées de Fourier, dont la FFT, servent essentiellement à faire du traitement de signaux. On a un signal donné dans un *domaine des temps*, à savoir une fonction qui associe un temps à une amplitude. L'analyse de Fourier permet d'exprimer le signal sous la forme d'une somme pondérée de sinusoides à décalage de phase et à fréquence variable. Les poids et phases associés aux fréquences caractérisent le signal dans le *domaine des fréquences*. Le traitement du signal est un domaine très riche qui a suscité une littérature de haut niveau, dont les notes du chapitres donnent un aperçu.

a) Polynômes

Un *polynôme* en x sur un corps algébrique K est la représentation d'une fonction $A(x)$ sous la forme d'une somme formelle :

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

Les valeurs a_0, a_1, \dots, a_{n-1} sont appelées **coefficients** du polynôme. Les coefficients appartiennent au corps K , qui est généralement le corps \mathbf{C} des nombres complexes. Un polynôme $A(x)$ est de **degré** k si son coefficient non nul de plus haut rang est a_k . Un entier strictement supérieur au degré d'un polynôme est un **majorant du degré** de ce polynôme. Le degré d'un polynôme de degré borné par n peut donc être n'importe quel entier compris entre 0 et $n - 1$ inclus.

Il est possible de définir une grande variété d'opérations sur les polynômes. Pour **l'addition des polynômes**, si $A(x)$ et $B(x)$ sont des polynômes de degrés majorés par n , leur **somme** est un polynôme $C(x)$, de degré également borné par n , tel que $C(x) = A(x) + B(x)$ pour tout x appartenant au corps sous-jacent. Autrement dit, si

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

et

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

alors

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,$$

où $c_j = a_j + b_j$ pour $j = 0, 1, \dots, n - 1$. Par exemple, si l'on a les polynômes $A(x) = 6x^3 + 7x^2 - 10x + 9$ et $B(x) = -2x^3 + 4x - 5$, alors $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Pour la **multiplication des polynômes**, si $A(x)$ et $B(x)$ sont des polynômes de degrés majorés par n , leur **produit** $C(x)$ est un polynôme, de degré borné par $2n - 1$, tel que $C(x) = A(x)B(x)$ pour tout x appartenant au corps sous-jacent. Vous avez sans doute déjà multiplié des polynômes, en multipliant chaque terme de $A(x)$ par chaque terme de $B(x)$ et en combinant les termes de puissances égales. Par exemple, on peut multiplier $A(x) = 6x^3 + 7x^2 - 10x + 9$ et $B(x) = -2x^3 + 4x - 5$ de la manière suivante :

$$\begin{array}{r} 6x^3 + 7x^2 - 10x + 9 \\ - 2x^3 \qquad \qquad + 4x - 5 \\ \hline - 30x^3 - 35x^2 + 50x - 45 \\ 24x^4 + 28x^3 - 40x^2 + 36x \\ - 12x^6 - 14x^5 + 20x^4 - 18x^3 \\ \hline - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{array}$$

Le produit $C(x)$ peut aussi s'exprimer sous la forme

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j, \tag{30.1}$$

où

$$c_j = \sum_{k=0}^j a_k b_{j-k}. \quad (30.2)$$

Notez que $\text{degré}(C) = \text{degré}(A) + \text{degré}(B)$, ce qui implique que, si A est un polynôme de degré borné par n_a et B un polynôme de degré borné par n_b , alors C est un polynôme de degré borné par $n_a + n_b - 1$. Comme un polynôme de degré borné par k est aussi un polynôme de degré borné par $k+1$, on dit normalement que le polynôme produit C est un polynôme de degré borné par $n_a + n_b$.

b) Présentation du chapitre

La section 30.1 montre deux façons de représenter les polynômes : la représentation par coefficients et la représentation par paires point-valeur. Les méthodes directes de multiplication de polynômes (équations (30.1) et (30.2)) demandent un temps $\Theta(n^2)$ quand les polynômes sont représentés par leurs coefficients, mais seulement $\Theta(n)$ seulement quand ils sont représentés par des paires point-valeur. On peut, toutefois, multiplier des polynômes, représentés par leurs coefficients, en un temps $\Theta(n \lg n)$, en effectuant une conversion entre les deux représentations. Pour comprendre ce mécanisme, il faut d'abord étudier les racines complexes de l'unité, ce que nous ferons à la section 30.2. Ensuite, nous utiliserons la FFT et sa réciproque, également décrite dans la section 30.2, pour effectuer les conversions. La section 30.3 montre comment implémenter efficacement la FFT tant dans le modèle sériel que dans le modèle parallèle.

Ce chapitre fait beaucoup appel aux nombres complexes et le symbole i sera employé exclusivement pour représenter $\sqrt{-1}$.

30.1 REPRÉSENTATION DES POLYNÔMES

La représentation par coefficients et la représentation par paires point-valeur sont, en un sens, équivalentes : tout polynôme représenté par un ensemble de paires point-valeur est associé à un et un seul polynôme représenté par des coefficients. Dans cette section, on introduit les deux représentations et on montre comment elles peuvent se combiner pour permettre de multiplier deux polynômes de degré borné par n en un temps $\Theta(n \lg n)$.

c) Représentation par coefficients

Une **représentation par coefficients** d'un polynôme $A(x) = \sum_{j=0}^{n-1} a_j x^j$ de degré borné par n est un vecteur de coefficients $a = (a_0, a_1, \dots, a_{n-1})$. Dans les équations matricielles de ce chapitre, on traitera généralement les vecteurs comme des vecteurs colonne.

La représentation par coefficients est commode pour certaines opérations sur les polynômes. Par exemple, l'opération **d'évaluation** du polynôme $A(x)$ en un point donné x_0 consiste à calculer la valeur de $A(x_0)$. L'évaluation nécessite un temps $\Theta(n)$ si l'on utilise la **règle de Horner** :

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))) \cdots).$$

De même, l'addition de deux polynômes représentés par les vecteurs de coefficients $a = (a_0, a_1, \dots, a_{n-1})$ et $b = (b_0, b_1, \dots, b_{n-1})$ prend un temps $\Theta(n)$: il suffit de calculer le vecteur de coefficients $c = (c_0, c_1, \dots, c_{n-1})$, où $c_j = a_j + b_j$ pour $j = 0, 1, \dots, n-1$.

Considérons à présent la multiplication de deux polynômes, de degré borné par n , $A(x)$ et $B(x)$ représentés par leurs coefficients. Si l'on utilise la méthode décrite aux équations (30.1) et (30.2), la multiplication des polynômes requiert $\Theta(n^2)$, puisque chaque coefficient du vecteur a doit être multiplié par chaque coefficient du vecteur b . La multiplication des polynômes représentés par leur coefficients paraît beaucoup plus complexe que l'évaluation d'un polynôme ou l'addition de polynômes. Le vecteur de coefficients résultant c , donné par l'équation (30.2), est également appelé **convolution** des vecteurs d'entrées a et b , ce qu'on note par $c = a \otimes b$. Comme la multiplication des polynômes et le calcul des convolutions sont des problèmes calculatoires fondamentaux d'une importance pratique considérable, ce chapitre insiste sur les algorithmes efficaces permettant de les résoudre.

d) Représentation par paires point-valeur

Une **représentation par paires point-valeur** d'un polynôme $A(x)$ de degré borné par n est un ensemble de n points du plan, représentés par leurs **coordonnées**

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

telles que les x_k sont tous distincts et que

$$y_k = A(x_k) \tag{30.3}$$

pour $k = 0, 1, \dots, n-1$. Un même polynôme possède plusieurs représentations par paires point-valeur, puisqu'un ensemble quelconque de n points distincts x_0, x_1, \dots, x_{n-1} peut servir de base à la représentation.

Le calcul d'une représentation par paires point-valeur d'un polynôme représenté par ses coefficients est simple dans son principe, puisqu'il suffit de choisir n abscisses distinctes x_0, x_1, \dots, x_{n-1} et d'évaluer ensuite $A(x_k)$ pour $k = 0, 1, \dots, n-1$. Avec la méthode de Horner, cette évaluation pour n points prend $\Theta(n^2)$. Nous verrons plus tard que, si l'on choisit intelligemment les x_k , le temps requis pour ce calcul peut être ramené à $\Theta(n \lg n)$.

L'inverse de l'évaluation, à savoir la détermination des coefficients d'un polynôme à partir d'une représentation par paires point-valeur, s'appelle **interpolation**. Le théorème suivant montre que l'interpolation est bien définie quand le degré du polynôme d'interpolation souhaité doit être borné par le nombre de paires point-valeur donné.

Théorème 30.1 (Unicité d'un polynôme d'interpolation) *Pour un ensemble $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ quelconque de n paires point-valeur telles que tous les x_k soient distincts, il existe un unique polynôme $A(x)$ de degré inférieur à n tel que $y_k = A(x_k)$ pour $k = 0, 1, \dots, n - 1$.*

Démonstration : La démonstration est basée sur l'existence de l'inverse d'une certaine matrice. L'équation (30.3) est équivalente à l'équation matricielle

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \quad (30.4)$$

La matrice de gauche est notée $V(x_0, x_1, \dots, x_{n-1})$ et est connue sous le nom de matrice de Vandermonde. D'après l'exercice 28.1.11, cette matrice a pour déterminant

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j),$$

et donc, d'après le théorème 28.5, elle est inversible (c'est-à-dire non singulière) si les x_k sont distincts. Donc, on peut trouver des coefficients a_j uniques à partir d'une quelconque représentation par paires point-valeur :

$$a = V(x_0, x_1, \dots, x_{n-1})^{-1}y.$$

□

La démonstration du théorème 30.1 décrit un algorithme d'interpolation basé sur la résolution de l'ensemble (30.4) d'équations linéaires. A l'aide des algorithmes de décomposition LU du chapitre 28, on peut résoudre ces équations en temps $O(n^3)$.

Un algorithme plus rapide pour l'interpolation sur n points est basé sur la **formule de Lagrange** :

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}. \quad (30.5)$$

On pourra vérifier que le membre droit de l'équation (30.5) est un polynôme de degré borné par n qui vérifie $A(x_i) = y_i$ pour tout i . L'exercice 30.1.5 vous demandera de calculer les coefficients de A à l'aide de la formule de Lagrange en temps $\Theta(n^2)$.

Ainsi, l'évaluation et l'interpolation de n points sont des opérations inverses bien définies qui permettent de passer de la représentation par coefficients d'un polynôme à sa représentation par paires point-valeur⁽¹⁾. Les algorithmes décrits ci-dessus pour ces problèmes s'exécutent en $\Theta(n^2)$.

(1) Il est bien connu que l'interpolation est un problème subtil quant à sa stabilité numérique. Bien que les approches décrites ici soient mathématiquement correctes, de petites différences dans les données ou des erreurs d'arrondi au cours du calcul peuvent provoquer de grandes différences dans le résultat.

La représentation par paires point-valeur est efficace pour de nombreuses opérations sur les polynômes. Pour l'addition, si $C(x) = A(x) + B(x)$, alors $C(x_k) = A(x_k) + B(x_k)$ pour tout x_k . Plus précisément, si l'on a une représentation par paires point-valeur pour A ,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\} ,$$

et pour B ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{n-1})\}$$

(notez que A et B sont évalués pour les *mêmes* n points), alors une représentation par paires point-valeur de C sera

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \dots, (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

Le temps nécessaire à l'addition de deux polynômes de degré borné par n , dans la représentation par paires point-valeur, est donc $\Theta(n)$.

La représentation par paires point-valeur est également pratique pour la multiplication des polynômes. Si $C(x) = A(x)B(x)$, alors $C(x_k) = A(x_k)B(x_k)$ pour tout x_k et l'on peut multiplier, point par point, une représentation par paires point-valeur de A par une représentation par paires point-valeur de B pour obtenir une représentation par paires point-valeur de C . Il faut, toutefois, résoudre le problème induit par $\text{degré}(C) = \text{degré}(A) + \text{degré}(B)$; si A et B sont de degré borné par n , alors C est de degré borné par $2n$. Une représentation par paires point-valeur normale de A et B se compose de n paires pour chaque polynôme. En multipliant ces paires deux à deux, on obtient n paires point-valeur; mais il faut $2n$ paires pour interpoler un polynôme unique C de degré borné par $2n$. (Voir exercice 30.1.4.) Il faut donc prendre des représentations « étendues » pour A et pour B , composées de $2n$ coordonnées chacune. Étant donnée une représentation étendue de A par paires point-valeur,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\} ,$$

et une représentation étendue correspondante pour B ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{2n-1}, y'_{2n-1})\} ,$$

C aura pour représentation étendue

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \dots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

Étant donnés deux polynômes représentés par des paires point-valeur étendues, on voit que le temps pris pour les multiplier et obtenir une représentation par valeurs du résultat est $\Theta(n)$; c'est très inférieur au temps nécessaire pour multiplier deux polynômes qui sont donnés par leurs coefficients.

Enfin, comment évaluer en un nouveau point un polynôme qui est représenté par des paires point-valeur? Pour ce problème, il n'existe apparemment pas de méthode plus simple que de calculer les coefficients du polynôme par interpolation, puis de l'évaluer au nouveau point.

e) *Multiplication rapide de polynômes donnés par leurs coefficients*

Est-il possible d'utiliser la méthode de multiplication à temps linéaire, employée pour des polynômes donnés par des paires point-valeur, pour multiplier des polynômes donnés par leurs coefficients ? La réponse est déterminée par notre capacité à convertir rapidement un polynôme de la représentation par coefficients vers la représentation par paires point-valeur (évaluation) et vice versa (interpolation).

On peut utiliser n'importe quel point comme point d'évaluation ; mais en choisissant soigneusement les points d'évaluation, on peut effectuer la conversion entre les deux représentations en seulement $\Theta(n \lg n)$. Comme nous le verrons à la section 30.2, si l'on choisit les « racines complexes de l'unité » comme points d'évaluation, on peut produire une représentation par paires point-valeur en prenant la transformée discrète de Fourier (DFT) d'un vecteur de coefficients. L'opération inverse, l'interpolation, peut s'effectuer en prenant la « DFT inverse » de paires point-valeur pour calculer un vecteur de coefficients. La section 30.2 montrera comment la transformée rapide de Fourier effectue la transformation discrète de Fourier et son inverse en $\Theta(n \lg n)$.

La figure 30.1 illustre graphiquement cette stratégie. Un petit détail à prendre en compte concerne les degrés. Le produit de deux polynômes de degré borné par n est un polynôme de degré borné par $2n$. Avant d'évaluer les polynômes A et B , il faut donc commencer par doubler leurs bornes de degré en ajoutant n coefficients de poids fort ayant la valeur 0. Comme les vecteurs ont $2n$ éléments, on utilise les « racines $(2n)$ ïèmes complexes de l'unité », qui sont représentées par les termes ω_{2n} sur la figure 30.1.

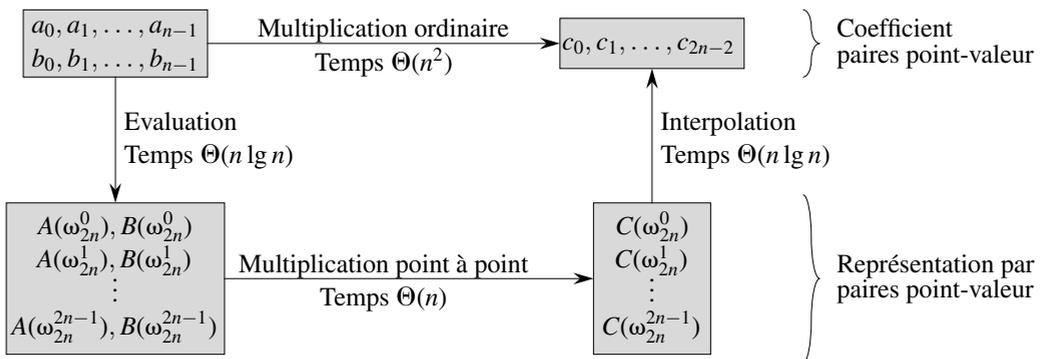


Figure 30.1 Schéma d'un processus efficace de multiplication de polynômes. Les représentations du haut sont par coefficients et celles du bas sont par paires point-valeur. Les flèches de la gauche vers la droite correspondent à l'opération de multiplication. Les termes ω_{2n} sont les racines $(2n)$ ïèmes complexes de l'unité.

Étant donnée la FFT, on a la procédure en $\Theta(n \lg n)$ suivante pour multiplier deux polynômes $A(x)$ et $B(x)$, de degré borné par n , où les représentations sont par coefficients en entrée comme en sortie. On suppose que n est une puissance de 2 ; cette

contrainte peut toujours être respectée en ajoutant des coefficients nuls dans les rangs supérieurs.

- 1) *Doublage de la borne* : On crée des représentations par coefficients de $A(x)$ et $B(x)$ pour en faire des polynômes de degré borné par $2n$, en ajoutant à chacun n coefficients nuls dans les rangs supérieurs.
- 2) *Évaluation* : On calcule les représentations par paires point-valeur de $A(x)$ et $B(x)$, de longueur $2n$, en appliquant deux fois la FFT d'ordre $2n$. Ces représentations contiennent les valeurs des deux polynômes pour les racines $(2n)$ ïèmes de l'unité.
- 3) *Multiplication point à point* : On calcule une représentation par paires point-valeur pour le polynôme $C(x) = A(x)B(x)$, en multipliant ces valeurs point à point. Cette représentation contient la valeur de $C(x)$ pour chaque racine $(2n)$ ïème de l'unité.
- 4) *Interpolation* : On crée la représentation par coefficients du polynôme $C(x)$ en appliquant une seule fois la FFT à $2n$ paires point-valeur pour calculer l'inverse de la transformée discrète de Fourier.

Les étapes (1) et (3) nécessitent $\Theta(n)$ et les étapes (2) et (4) $\Theta(n \lg n)$. Donc, une fois que nous aurons montré comment utiliser la FFT, nous aurons prouvé le théorème suivant :

Théorème 30.2 *Le produit de deux polynômes de degré borné par n peut être calculé en $\Theta(n \lg n)$, si l'entrée et la sortie sont représentées par leur coefficients.*

Exercices

30.1.1 Multiplier les polynômes $A(x) = 7x^3 - x^2 + x - 10$ et $B(x) = 8x^3 - 6x + 3$ à l'aide des équations (30.1) et (30.2).

30.1.2 L'évaluation d'un polynôme $A(x)$ de degré borné par n en un point donné x_0 peut aussi être effectuée en divisant $A(x)$ par le polynôme $(x - x_0)$ pour obtenir un polynôme quotient $q(x)$ de degré borné par $n - 1$ et un reste r , tel que

$$A(x) = q(x)(x - x_0) + r.$$

On voit clairement que $A(x_0) = r$. Montrer comment calculer le reste r et les coefficients de $q(x)$ en temps $\Theta(n)$ à partir de x_0 et des coefficients de A .

30.1.3 Dédurre une représentation par paires point-valeur de $A^{\text{mir}}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ à partir d'une représentation par paires point-valeur de $A(x) = \sum_{j=0}^{n-1} a_jx^j$, en supposant qu'aucun des points n'est 0.

30.1.4 Prouver qu'il faut n paires point-valeur distinctes pour définir sans ambiguïté un polynôme de degré borné par n ; en d'autres termes, si l'on a moins de n paires point-valeur, elles ne peuvent pas définir un polynôme unique de degré borné par n . (*conseil* : En utilisant le théorème 30.1, que peut-on dire d'un ensemble de $n - 1$ paires point-valeur auquel on ajoute une paire supplémentaire choisie arbitrairement ?)

30.1.5 Montrer comment utiliser l'équation (30.5) pour interpoler en $\Theta(n^2)$. (*conseil* : Commencer par calculer la représentation par coefficients du polynôme $\prod_j (x - x_j)$, puis diviser par $(x - x_k)$ comme numérateur de chaque terme; voir exercice 30.1.2.) Chacun des n dénominateurs devra être calculé en temps $O(n)$

30.1.6 Expliquer l'inconvénient de l'approche « directe » pour la division de polynômes basée sur la représentation par paires point-valeur. Étudier séparément le cas où la division tombe juste et celui où le reste n'est pas nul.

30.1.7 On considère deux ensembles A et B , contenant chacun n entiers compris entre 0 et $10n$. On souhaite calculer la *somme cartésienne* de A et B , définie par

$$C = \{x + y : x \in A \text{ et } y \in B\} .$$

Notez que les entiers de C sont compris entre 0 et $20n$. On veut trouver les éléments de C et le nombre de fois que chaque élément de C est produit par une somme d'éléments de A et B . Montrer qu'on peut résoudre ce problème en temps $O(n \lg n)$. (*conseil* : Représenter A et B sous la forme de polynômes de degré $10n$ au plus.)

30.2 TRANSFORMÉE DISCRÈTE DE FOURIER ET TRANSFORMÉE RAPIDE DE FOURIER

À la section 30.1, nous affirmions que, si l'on utilisait les racines complexes de l'unité, on pouvait évaluer et interpoler les polynômes en temps $\Theta(n \lg n)$. Dans cette section, nous définissons les racines complexes de l'unité et étudions leurs propriétés, nous définissons la transformée discrète de Fourier (DFT) et montrons comment la FFT peut calculer la DFT et son inverse en seulement $\Theta(n \lg n)$.

a) Racines complexes de l'unité

Une *racine nième complexe de l'unité* est un nombre complexe ω tel que

$$\omega^n = 1 .$$

Il existe exactement n racines nièmes complexes de l'unité ; $e^{2\pi i k/n}$ pour $k = 0, 1, \dots, n - 1$. Pour interpréter cette formule, on utilise la définition de l'exponentielle d'un nombre complexe :

$$e^{iu} = \cos(u) + i \sin(u) .$$

La figure 30.2 montre que les n racines complexes de l'unité sont espacées de façon régulière autour du cercle de rayon unitaire centré à l'origine du plan complexe. La valeur

$$\omega_n = e^{2\pi i/n} \quad (30.6)$$

est appelée *racine nième principale de l'unité* ; toutes les autres racines complexes de l'unité sont des puissances de ω_n .⁽²⁾

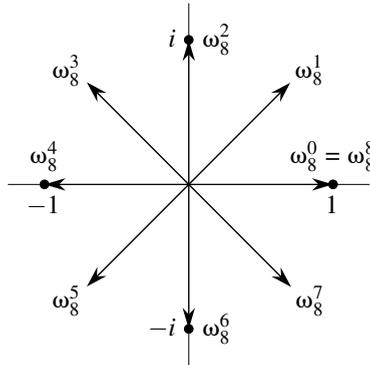


Figure 30.2 Les valeurs de $\omega_8^0, \omega_8^1, \dots, \omega_8^7$ dans le plan complexe, où $\omega_8 = e^{2\pi i/8}$ est la racine 8ième principale de l'unité.

Les n racines nièmes de l'unité,

$$\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1},$$

forment un groupe pour la multiplication (voir section 31.3). Ce groupe a la même structure que le groupe additif $(\mathbf{Z}_n, +)$ modulo n , puisque $\omega_n^n = \omega_n^0 = 1$ implique $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. De même, $\omega_n^{-1} = \omega_n^{n-1}$. Les lemmes suivants donnent les propriétés essentielles des racines nièmes de l'unité.

Lemme 30.3 (Lemme de l'annulation) *Quels que soient les entiers $n \geq 0$, $k \geq 0$ et $d > 0$,*

$$\omega_{dn}^{dk} = \omega_n^k. \quad (30.7)$$

Démonstration : Le lemme se déduit directement de l'équation (30.6), puisque

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn} \right)^{dk} \\ &= \left(e^{2\pi i/n} \right)^k \\ &= \omega_n^k. \end{aligned}$$

□

(2) Maints auteurs définissent ω_n différemment : $\omega_n = e^{-2\pi i/n}$. Cette autre définition tend à s'imposer dans les applications du traitement du signal. Les mathématiques sous-jacentes sont essentiellement les mêmes, quelle que soit la définition de ω_n .

Corollaire 30.4 Pour tout entier pair $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Démonstration : La démonstration est laissée en exercice (voir exercice 30.2.1). \square

Lemme 30.5 (Lemme de la bipartition) Si $n > 0$ est pair, les carrés des n racines n èmes de l'unité sont les $n/2$ racines $(n/2)$ èmes de l'unité.

Démonstration : D'après le lemme de l'annulation, on a $(\omega_n^k)^2 = \omega_{n/2}^k$, pour tout entier non négatif k . Notez que, si l'on élève au carré toutes les racines n èmes de l'unité, chaque racine $(n/2)$ ème de l'unité est obtenue exactement deux fois, puisque

$$\begin{aligned} (\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k} \omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 . \end{aligned}$$

Donc, ω_n^k et $\omega_n^{k+n/2}$ ont le même carré. Cette propriété peut également être démontrée à l'aide du corollaire 30.4, puisque $\omega_n^{n/2} = -1$ implique $\omega_n^{k+n/2} = -\omega_n^k$ et donc $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. \square

Comme nous le verrons, le lemme de la bipartition est essentiel pour notre approche diviser-pour-régner de la conversion entre représentation par coefficients et représentation par paires point-valeur des polynômes, car elle garantit que les sous-problèmes récursifs seront deux fois moins grands.

Lemme 30.6 (Lemme de la sommation) Pour tout entier $n \geq 1$ et tout entier k non nul et non divisible par n ,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 .$$

Démonstration : L'équation (A.5) s'appliquant aussi aux valeurs complexes, on a donc

$$\begin{aligned} \sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} = 0 . \end{aligned}$$

On impose à k de ne pas être divisible par n pour assurer que le dénominateur n'est pas nul. En effet, $\omega_n^k = 1$ seulement quand k est divisible par n . \square

b) Transformée discrète de Fourier

On se souvient que notre objectif est d'évaluer un polynôme

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

de degré borné par n sur les points $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$ (c'est-à-dire pour les n racines n èmes de l'unité) ⁽³⁾. On peut supposer que n est une puissance de 2 sans nuire à la généralité, puisqu'une borne de degré donnée peut toujours être augmentée (il suffit d'ajouter des coefficients de valeur nulle). ⁽⁴⁾ On suppose que A est donné par ses coefficients : $a = (a_0, a_1, \dots, a_{n-1})$. On définit les résultats y_k , pour $k = 0, 1, \dots, n-1$, par

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \end{aligned} \quad (30.8)$$

Le vecteur $y = (y_0, y_1, \dots, y_{n-1})$ est la **transformée discrète de Fourier (DFT)** du vecteur de coefficients $a = (a_0, a_1, \dots, a_{n-1})$. On écrit également $y = \text{DFT}_n(a)$.

c) Transformée rapide de Fourier

En utilisant une méthode connue sous le nom de **Transformée rapide de Fourier (FFT)**, qui tire parti des propriétés particulières des racines complexes de l'unité, on peut calculer $\text{DFT}_n(a)$ en temps $\Theta(n \lg n)$, contrairement au temps $\Theta(n^2)$ de la méthode directe.

La méthode FFT fait appel à une stratégie diviser-pour-régner, en se servant séparément des coefficients d'indice pair et des coefficients d'indice impair de $A(x)$ pour définir les deux nouveaux polynômes $A^{[0]}(x)$ et $A^{[1]}(x)$ de degré majoré par $n/2$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}. \end{aligned}$$

Notez que $A^{[0]}$ contient tous les coefficients d'indice pair de A (la représentation binaire de l'indice se termine par 0) et que $A^{[1]}$ contient tous les coefficients d'indice impair (la représentation binaire de l'indice se termine par 1). Il s'ensuit que

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2), \quad (30.9)$$

(3) La longueur n est en réalité ce que nous appelions $2n$ à la section 30.1, puisque la borne du degré des deux polynômes est doublée avant l'évaluation. Dans le cas d'une multiplication de polynômes, on travaille donc en réalité avec les racines $(2n)$ èmes de l'unité.

(4) Quand on emploie la FFT pour faire du traitement du signal, il est généralement déconseillé d'ajouter des coefficients nuls pour arriver à une taille qui soit une puissance de 2, car cela tend à introduire des parasites haute fréquence. Une technique utilisée dans le traitement du signal pour cadrer sur une puissance de 2 consiste à utiliser un **miroir**. Si n' désigne la plus petite puissance entière de 2 qui est plus grande que n , une façon de miroiter consiste à faire les affectations $a_{n'+j} = a_{n-j-2}$ pour $j = 0, 1, \dots, n' - n - 1$.

de sorte que le problème consistant à évaluer $A(x)$ en $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ se ramène à

1) évaluer les polynômes de borne $n/2$ $A^{[0]}(x)$ et $A^{[1]}(x)$ aux points

$$(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2, \quad (30.10)$$

puis

2) combiner les résultats selon le modèle de l'équation (30.9).

D'après le lemme de la bipartition, la liste des valeurs (30.10) n'est pas composée de n valeurs distinctes, mais seulement des $n/2$ racines $(n/2)$ ïèmes de l'unité, chaque racine apparaissant exactement deux fois. Ainsi, les polynômes $A^{[0]}$ et $A^{[1]}$ de borne $n/2$ sont évalués récursivement pour les $n/2$ racines $(n/2)$ ïèmes complexes de l'unité. Ces sous-problèmes ont exactement la même forme que le problème initial, mais ils sont moitié moins grands. Nous avons maintenant réussi à diviser un calcul DFT_n à n éléments en deux calculs $DFT_{n/2}$ à $n/2$ éléments. Cette décomposition est la base de l'algorithme récursif suivant, qui calcule la transformée discrète de Fourier d'un vecteur à n éléments $a = (a_0, a_1, \dots, a_{n-1})$, où n est une puissance de 2.

FFT-RÉCURSIVE(a)

```

1   $n \leftarrow \text{longueur}[a]$             $\triangleright n$  est une puissance de 2.
2  si  $n = 1$ 
3    alors retourner  $a$ 
4   $\omega_n \leftarrow e^{2\pi i/n}$ 
5   $\omega \leftarrow 1$ 
6   $a^{[0]} \leftarrow (a_0, a_2, \dots, a_{n-2})$ 
7   $a^{[1]} \leftarrow (a_1, a_3, \dots, a_{n-1})$ 
8   $y^{[0]} \leftarrow \text{FFT-RÉCURSIVE}(a^{[0]})$ 
9   $y^{[1]} \leftarrow \text{FFT-RÉCURSIVE}(a^{[1]})$ 
10 pour  $k \leftarrow 0$  à  $n/2 - 1$ 
11   faire  $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$ 
12          $y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$ 
13          $\omega \leftarrow \omega \omega_n$ 
14 retourner  $y$             $\triangleright y$  est supposé être un vecteur colonne.
```

La procédure FFT-RÉCURSIVE fonctionne de la manière suivante. Les lignes 2–3 représentent la base de la récursivité ; la transformée discrète de Fourier d'un seul élément est l'élément lui-même, puisque dans ce cas

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0. \end{aligned}$$

Les lignes 6–7 définissent les vecteurs de coefficients des polynômes $A^{[0]}$ et $A^{[1]}$. Les lignes 4, 5 et 13 garantissent que ω est mis à jour de telle sorte qu'à chaque exécution des lignes 11–12, $\omega = \omega_n^k$. (Maintenir à jour une valeur de ω d'une itération à l'autre

permet de gagner du temps par rapport à un calcul ω_n^k *ex nihilo* à chaque passage dans la boucle **pour**.) Les lignes 8–9 calculent récursivement les $\text{DFT}_{n/2}$, en effectuant, pour $k = 0, 1, \dots, n/2 - 1$, les affectations

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k), \end{aligned}$$

ou, puisque $\omega_{n/2}^k = \omega_n^{2k}$ d'après le lemme de l'annulation,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}), \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}). \end{aligned}$$

Les lignes 11–12 combinent les résultats des calculs récursifs de $\text{DFT}_{n/2}$. Pour $y_0, y_1, \dots, y_{n/2-1}$, la ligne 11 donne

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \quad (\text{d'après l'équation (30.9)}). \end{aligned}$$

Pour $y_{n/2}, y_{n/2+1}, \dots, y_{n-1}$, en faisant $k = 0, 1, \dots, n/2 - 1$, la ligne 12 donne

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \quad (\text{car } \omega_n^{k+(n/2)} = -\omega_n^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \quad (\text{car } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)}) \quad (\text{d'après l'équation (30.9)}). \end{aligned}$$

Donc, le vecteur y retourné par FFT-RÉCURSIVE est bien la transformée discrète de Fourier du vecteur d'entrée a .

Dans la boucle **pour** des lignes 10–13, chaque valeur $y_k^{[1]}$ est multipliée par ω_n^k , pour $k = 0, 1, \dots, n/2 - 1$. Le produit est à la fois ajouté et soustrait à $y_k^{[0]}$. Comme chaque facteur ω_n^k est utilisé dans sa forme positive et dans sa forme négative, les facteurs ω_n^k sont appelés *facteurs tournants*.

Pour déterminer le temps d'exécution de la procédure FFT-RÉCURSIVE, on remarque que, si l'on omet les appels récursifs, chaque appel prend un temps $\Theta(n)$, où n est la longueur du vecteur d'entrée. La récurrence pour le temps d'exécution est donc

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

Donc, un polynôme de degré borné par n peut être évalué sur les racines n èmes de l'unité en temps $\Theta(n \lg n)$ à l'aide la transformée rapide de Fourier.

d) Interpolation aux racines complexes de l'unité

Complétons à présent le mécanisme de la multiplication des polynômes en montrant comment interpoler les racines complexes de l'unité par un polynôme, ce qui nous permettra de passer de la représentation par paires point-valeur à la représentation par coefficients. On interpole en écrivant la transformée discrète de Fourier sous la forme d'une équation matricielle, puis en s'intéressant à la forme de la matrice inverse.

D'après l'équation (30.4), on peut écrire la transformée discrète de Fourier comme le produit de matrice $y = V_n a$, où V_n est une matrice de Vandermonde contenant les puissances appropriées de ω_n :

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

La composante (k, j) de V_n est égale à ω_n^{kj} , pour $j, k = 0, 1, \dots, n - 1$, et les exposants des composantes de V_n forment une table de multiplication.

Pour l'opération inverse, que l'on écrit $a = \text{DFT}_n^{-1}(y)$, on peut procéder en multipliant y par la matrice V_n^{-1} , inverse de V_n .

Théorème 30.7 Pour $j, k = 0, 1, \dots, n - 1$, la composante (j, k) de V_n^{-1} est ω_n^{-kj}/n .

Démonstration : On montre que $V_n^{-1}V_n = I_n$, matrice identité $n \times n$. Considérons la composante (j, j') de $V_n^{-1}V_n$:

$$\begin{aligned} [V_n^{-1}V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n . \end{aligned}$$

Cette sommation vaut 1 si $j' = j$ et 0 dans le cas contraire, d'après le lemme de sommation (lemme 30.6). Notez qu'on s'appuie sur le fait que $-(n - 1) \leq j' - j \leq n - 1$, de sorte que $j' - j$ n'est pas divisible par n , pour pouvoir appliquer le lemme de sommation. □

Connaissant la matrice inverse V_n^{-1} , $\text{DFT}_n^{-1}(y)$ est donnée par

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

pour $j = 0, 1, \dots, n - 1$. En comparant les équations (30.8) et (30.11), on voit que, en modifiant l'algorithme FFT de façon à intervertir les rôles de a et y , remplacer ω_n par

ω_n^{-1} et diviser chaque élément du résultat par n , on calcule l'inverse de la transformée discrète de Fourier (voir exercice 30.2.4). Donc, DFT_n^{-1} peut également être calculée en temps $\Theta(n \lg n)$.

Donc, en utilisant la FFT et son inverse, il est possible de faire passer un polynôme, de degré borné par n , de sa représentation par coefficients à sa représentation par paires point-valeur, et réciproquement, en temps $\Theta(n \lg n)$. Dans le cas de la multiplication des polynômes, nous avons montré la chose suivante :

Théorème 30.8 (Théorème de convolution) *Pour deux vecteurs quelconques a et b de longueur n , où n est une puissance de 2,*

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)),$$

où les vecteurs a et b sont complétés avec des 0 jusqu'à la longueur $2n$ et où \cdot représente le produit composante par composante des deux vecteurs à $2n$ éléments.

Exercices

30.2.1 Démontrer le corollaire 30.4.

30.2.2 Calculer la transformée discrète de Fourier du vecteur $(0, 1, 2, 3)$.

30.2.3 Faire l'exercice 30.1.1 en utilisant le modèle temporel $\Theta(n \lg n)$.

30.2.4 Écrire un pseudo code permettant de calculer DFT_n^{-1} en temps $\Theta(n \lg n)$.

30.2.5 Décrire la généralisation de la procédure FFT au cas où n est une puissance de 3. Donner une récurrence pour le temps d'exécution et la résoudre.

30.2.6 * Supposons qu'au lieu d'effectuer une FFT à n éléments sur le corps des nombres complexes (pour n pair), on utilise l'anneau \mathbf{Z}_m des entiers modulo m , où $m = 2^{m/2} + 1$ et t est un entier positif arbitraire. On utilise $w = 2^t$ à la place de ω_n comme racine n ème principale de l'unité, modulo m . Démontrer que la transformée discrète de Fourier et son inverse sont bien définies dans ce système.

30.2.7 Étant donnée une liste de valeurs z_0, z_1, \dots, z_{n-1} (avec répétitions possibles), montrer comment trouver les coefficients d'un polynôme $P(x)$, de degré borné par n , qui s'annule uniquement en z_0, z_1, \dots, z_{n-1} (avec répétitions éventuelles). Votre procédure devra s'exécuter en temps $O(n \lg^2 n)$. (conseil : Le polynôme $P(x)$ a un zéro en z_j si et seulement si $P(x)$ est un multiple de $(x - z_j)$.)

30.2.8 * La *transformée chirp* d'un vecteur $a = (a_0, a_1, \dots, a_{n-1})$ est le vecteur $y = (y_0, y_1, \dots, y_{n-1})$, où $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ et z est un nombre complexe quelconque. La transformée discrète de Fourier est donc un cas particulier de la transformée chirp, obtenue en prenant $z = \omega_n$. Démontrer que la transformée chirp peut être évaluée en temps $O(n \lg n)$ pour un nombre complexe z quelconque. (*Conseil* : Utiliser l'équation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2}) (z^{-(k-j)^2/2})$$

pour voir la transformée chirp comme une convolution.)

30.3 IMPLÉMENTATIONS EFFICACES DE LA FFT

Les applications pratiques de la transformée discrète de Fourier, comme le traitement du signal, nécessitant la vitesse la plus grande possible, cette section se propose d'étudier deux implémentations efficaces de la FFT. D'abord, on commence par examiner une version itérative de l'algorithme de la FFT qui s'exécute en $\Theta(n \lg n)$ mais possède une constante cachée dans la notation Θ plus petite que l'implémentation récursive vue à la section 30.2. Ensuite, nous utiliserons les connaissances qui nous ont conduits à l'implémentation itérative pour élaborer un circuit FFT parallèle efficace.

a) Une implémentation itérative de la FFT

Commençons par remarquer que la boucle **pour** des lignes 10–13 de FFT-RÉCURSIVE implique le calcul à deux reprises de la valeur $\omega_n^k y_k^{[1]}$. En terminologie des compilateurs, ceci s'appelle une *sous-expression commune*. On peut modifier la boucle pour la calculer une seule fois, en la stockant dans une variable temporaire t .

```

pour  $k \leftarrow 0$  à  $n/2 - 1$ 
  faire  $t \leftarrow \omega y_k^{[1]}$ 
         $y_k \leftarrow y_k^{[0]} + t$ 
         $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$ 
   $\omega \leftarrow \omega \omega_n$ 

```

L'action de cette boucle qui multiplie le facteur tournant $\omega = \omega_n^k$ par $y_k^{[1]}$, stocke le produit dans t , puis ajoute et soustrait t de $y_k^{[0]}$, est connue sous le nom de *opération papillon* et est montrée schématiquement à la figure 30.3.

Voici à présent comment rendre la structure de l'algorithme FFT itérative et non plus récursive. À la figure 30.4, nous avons représenté dans une structure arborescente les vecteurs d'entrée des appels récursifs d'une invocation de FFT-RÉCURSIVE, l'appel initial s'effectuant pour $n = 8$. L'arbre contient un nœud pour chaque appel à la procédure, étiqueté par le vecteur d'entrée correspondant. Chaque invocation de FFT-RÉCURSIVE effectue deux appels récursifs, à moins qu'elle reçoive en argument un

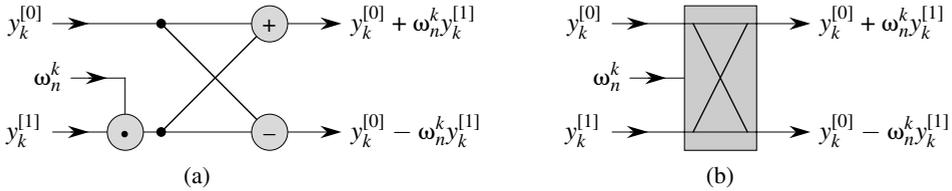


Figure 30.3 Une opération papillon. (a) Les deux valeurs d’entrée arrivent de la gauche, le facteur tournant ω_n^k est multiplié par $y_k^{[1]}$, puis la somme et la différence sont produites sur la droite. (b) Dessin simplifié d’une opération papillon. Nous utiliserons cette représentation dans un circuit FFT parallèle.

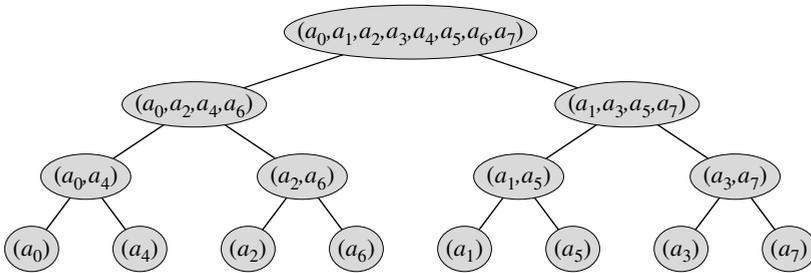


Figure 30.4 L’arborescence des vecteurs d’entrées pour les appels récursifs de la procédure FFT-RÉCURSIVE. Le premier appel se fait pour $n = 8$.

vecteur à 1 élément. Le premier appel est figuré par le fils gauche, et le second par le fils droit.

En observant cette arborescence, on remarque que, si l’on pouvait réorganiser les éléments du vecteur initial a dans leur ordre d’apparition aux feuilles, il serait possible de simuler l’exécution de la procédure FFT-RÉCURSIVE comme suit. D’abord, on prend les éléments deux par deux, on calcule la transformée discrète de Fourier de chaque paire à l’aide d’une opération papillon et on remplace la paire par sa transformée discrète de Fourier. Le vecteur contient alors $n/2$ DFT à 2 éléments. Ensuite, on prend ces $n/2$ DFT deux par deux et on calcule la transformée discrète de Fourier des quatre éléments de vecteurs qui viennent de l’exécution de deux opérations papillon, en remplaçant les deux DFT à 2 éléments par une DFT à 4 éléments. Le vecteur contient maintenant $n/4$ DFT à 4 éléments. On continue de cette manière jusqu’à ce que le vecteur contienne deux DFT à $(n/2)$ éléments, qu’on peut alors combiner à l’aide de $n/2$ opérations papillons pour donner la DFT finale à n éléments.

Pour transformer cette observation en code, on utilise un tableau $A[0 \dots n - 1]$ qui contient initialement les éléments du vecteur d’entrée a dans l’ordre où ils apparaissent dans les feuilles de l’arborescence de la figure 30.4. (On montrera plus tard comment déterminer cet ordre, connu sous le nom de permutation à inversion de bits). La combinaison devant être effectuée à chaque niveau de l’arborescence, on introduit une variable s pour compter les niveaux, qui vont de 1 (en bas, quand on combine des

paire pour former des DFT à 2 éléments) à $\lg n$ (en haut, quand on combine deux DFT à $(n/2)$ éléments pour produire le résultat final). L'algorithme a donc la structure suivante :

```

1  pour  $s \leftarrow 1$  à  $\lg n$ 
2      faire pour  $k \leftarrow 0$  à  $n - 1$  avec un pas  $2^s$ 
3          faire combiner les deux DFT à  $2^{s-1}$  éléments
                 $A[k \dots k + 2^{s-1} - 1]$  et  $A[k + 2^{s-1} \dots k + 2^s - 1]$ 
                pour en faire une seule DFT à  $2^s$  éléments  $A[k \dots k + 2^s - 1]$ 

```

On peut exprimer le corps de la boucle (ligne 3) à l'aide d'un pseudo code plus détaillé. On recopie la boucle **pour** de la procédure FFT-RÉCURSIVE, en identifiant $y^{[0]}$ avec $A[k \dots k + 2^{s-1} - 1]$ et $y^{[1]}$ avec $A[k + 2^{s-1} \dots k + 2^s - 1]$. Le facteur tournant utilisé dans chaque opération papillon dépend de la valeur de s ; c'est une puissance de ω_m , où $m = 2^s$. (La variable m n'est introduite que pour une meilleure lisibilité.) On ajoute une variable temporaire u qui nous permet d'effectuer l'opération papillon sur place. Lorsqu'on remplace la ligne 3 de la structure globale par le corps de la boucle, on obtient le pseudo code suivant qui forme la base de l'implémentation parallèle que nous présenterons plus loin. Le code commence par appeler la procédure auxiliaire COPIE-INVERSION-BITS(a, A) pour copier le vecteur a dans le tableau A dans l'ordre initial dans lequel il nous faut les valeurs.

FFT-ITÉRATIVE(a)

```

1  COPIE-INVERSION-BITS( $a, A$ )
2   $n \leftarrow \text{longueur}[a]$            $\triangleright n$  est une puissance de 2.
3  pour  $s \leftarrow 1$  à  $\lg n$ 
4      faire  $m \leftarrow 2^s$ 
5           $\omega_m \leftarrow e^{2\pi i/m}$ 
6          pour  $k \leftarrow 0$  à  $n - 1$  avec un pas  $m$ 
7              faire  $\omega \leftarrow 1$ 
8                  pour  $j \leftarrow 0$  à  $m/2 - 1$ 
9                      faire  $t \leftarrow \omega A[k + j + m/2]$ 
10                          $u \leftarrow A[k + j]$ 
11                          $A[k + j] \leftarrow u + t$ 
12                          $A[k + j + m/2] \leftarrow u - t$ 
13                          $\omega \leftarrow \omega \omega_m$ 

```

Comment COPIE-INVERSION-BITS place-t-elle les éléments du vecteur d'entrée a dans le bon ordre dans le tableau A ? L'ordre dans lequel les feuilles apparaissent sur la figure 30.4 est une *permutation à inversion de bits*. Autrement dit, $\text{inv}(k)$ désignant l'entier de $\lg n$ bits formé en inversant les bits de la représentation binaire de k , on souhaite placer l'élément a_k du vecteur à la position $A[\text{inv}(k)]$ dans le tableau. Sur la figure 30.4, par exemple, les feuilles apparaissent dans l'ordre 0, 4, 2, 6, 1, 5, 3, 7 ; cette séquence, exprimée en binaire, est

000, 100, 010, 110, 001, 101, 011, 111 et si l'on renverse les bits de chaque valeur, on obtient la séquence 000, 001, 010, 011, 100, 101, 110, 111. Pour comprendre pourquoi cette transformation est souhaitable dans le cas général, on remarque qu'au niveau supérieur de l'arborescence, les indices dont le bit d'ordre inférieur est 0 sont placés dans la sous-arborescence gauche et ceux dont le bit d'ordre inférieur est égal à 1 sont placés dans la sous-arborescence de droite. En supprimant le bit de poids faible à chaque niveau, on continue ce processus tout le long de l'arbre, pour obtenir au niveau des feuilles l'ordre donné par la permutation à inversion de bits.

La fonction $\text{inv}(k)$ étant facile à calculer, la procédure COPIE-INVERSION-BITS peut être écrite de la manière suivante.

COPIE-INVERSION-BITS(a, A)

```

1   $n \leftarrow \text{longueur}[a]$ 
2  pour  $k \leftarrow 0$  à  $n - 1$ 
3      faire  $A[\text{inv}(k)] \leftarrow a_k$ 
```

L'implémentation de la FFT itérative s'exécute en $\Theta(n \lg n)$. L'appel à COPIE-INVERSION-BITS(a, A) s'exécute à coup sûr en $O(n \lg n)$, puisqu'on itère n fois et qu'on peut inverser un entier entre 0 et $n - 1$, à $\lg n$ bits, en $O(\lg n)$. (En pratique, on connaît en général à l'avance la valeur initiale de n , ce qui permet de coder une table de correspondance entre k et $\text{inv}(k)$. De ce fait, COPIE-INVERSION-BITS peut s'exécuter en $\Theta(n)$, avec une constante cachée faible. On pourrait également utiliser l'astucieux mécanisme amorti du compteur binaire inverse décrit au problème 17.1.) Pour finir de prouver que FFT-ITÉRATIVE s'exécute en $\Theta(n \lg n)$, on va montrer que $L(n)$, nombre d'exécutions de la boucle interne (lignes 8–13), est $\Theta(n \lg n)$. La boucle **pour** des lignes 6–13 est exécutée $n/m = n/2^s$ fois pour chaque valeur de s , et la boucle la plus interne des lignes 8–13 est effectuée $m/2 = 2^{s-1}$ fois. D'où

$$\begin{aligned}
 L(n) &= \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} \\
 &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
 &= \sum_{s=1}^{\lg n} \frac{n}{2} \\
 &= \Theta(n \lg n) .
 \end{aligned}$$

b) Un circuit FFT parallèle

Il est possible d'exploiter une grande partie des propriétés qui nous ont permis d'implémenter un algorithme FFT itératif efficace, pour produire un algorithme parallèle efficace pour la FFT.

Nous représenterons l'algorithme parallèle FFT par un circuit qui rappelle beaucoup les réseaux de comparaison du chapitre 27. Au lieu de comparateurs, le circuit FFT emploie des opérations papillon, comme indiqué sur la figure 30.3(b). La notion de profondeur, introduite au chapitre 27, s'applique encore ici. Le circuit FFT-PARALLÈLE qui calcule la FFT sur n entrées est montré à la figure 30.5 pour $n = 8$. Le circuit commence par une permutation des entrées à base d'inversion de bits, suivie de $\lg n$ paliers, chacun étant constitué de $n/2$ opérations papillon exécutées en parallèle. La profondeur du circuit est donc $\Theta(\lg n)$.

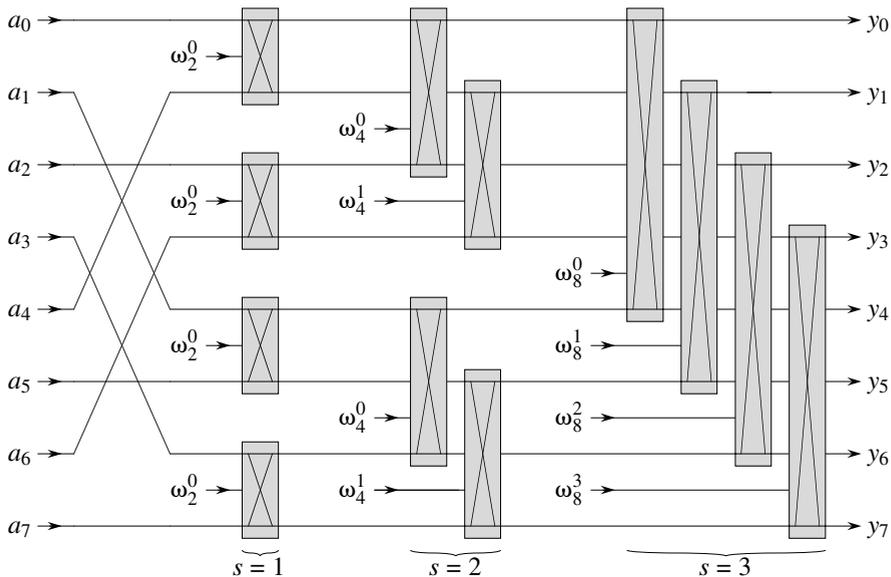


Figure 30.5 Un circuit FFT-PARALLÈLE qui calcule la FFT, ici sur $n = 8$ entrées. Chaque opération papillon prend en entrée les valeurs sur deux fils, plus un facteur tournant, et produit en sortie les valeurs sur deux fils. Les étapes des papillons sont étiquetées de façon à refléter les itérations de la boucle la plus externe de la procédure FFT-ITÉRATIVE. Seuls les fils du haut et du bas qui passent à travers un papillon interagissent avec lui ; les fils qui passent au milieu d'un papillon n'affectent pas le papillon, et réciproquement le papillon ne modifie pas leurs valeurs. Par exemple, le papillon du haut de l'étape 2 n'a rien à faire du fil 1 (fil dont la sortie est étiquetée y_1) ; ses entrées et sorties sont uniquement sur les fils 0 et 2 (intitulés y_0 et y_2). Une FFT à n entrées peut être calculée sur une profondeur en $\Theta(\lg n)$ avec $\Theta(n \lg n)$ opérations papillon.

La partie la plus à gauche du circuit FFT-PARALLÈLE effectue la permutation par inversion de bits, et le reste imite la procédure itérative FFT-ITÉRATIVE. Comme chaque itération de la boucle **pour** la plus externe effectue $n/2$ opérations papillon

indépendantes, le circuit les fait en parallèle. La valeur de s dans chaque itération dans FFT-ITÉRATIVE correspond à un palier de papillons montré sur la figure 30.5. A l'intérieur d'un palier s , pour $s = 1, 2, \dots, \lg n$, on a $n/2^s$ groupes de papillons (correspondant à chaque valeur de k dans FFT-ITÉRATIVE), avec 2^{s-1} papillons par groupe (correspondant à chaque valeur de j dans FFT-ITÉRATIVE). Les papillons montrés sur la figure 30.5 correspondent aux opérations papillon de la boucle la plus interne (lignes 9–12 de FFT-ITÉRATIVE). Notez aussi que les facteurs tournants utilisés dans les papillons correspondent à ceux utilisés dans FFT-ITÉRATIVE : au palier s , on utilise $\omega_m^0, \omega_m^1, \dots, \omega_m^{m/2-1}$, où $m = 2^s$.

Exercices

30.3.1 Montrer comment FFT-ITÉRATIVE calcule la transformée discrète de Fourier du vecteur d'entrée $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3.2 Montrer comment implémenter un algorithme FFT qui effectue la permutation par inversion de bits à la fin, et non au début, du calcul. (*conseil* : Considérer la transformée discrète de Fourier inverse.)

30.3.3 Combien de fois FFT-ITÉRATIVE calcule-t-elle des facteurs tournants à chaque étape ? Réécrire FFT-ITÉRATIVE pour qu'elle ne calcule des facteurs tournants que 2^{s-1} fois à l'étape stage s .

30.3.4 * Supposons que les additionneurs internes aux opérations papillon du circuit FFT aient parfois un comportement erratique, tel qu'ils produisent toujours un résultat nul, indépendamment de la valeurs de leurs entrées. Supposons qu'un additionneur et un seul ait ce type de comportement, mais qu'on ne sache pas lequel. Décrire une façon d'identifier l'additionneur fautif en fournissant des entrées au circuit FFT global et en observant les résultats. Quelle est l'efficacité de cette méthode ?

PROBLÈMES

30.1. Multiplication diviser-pour-régner

- Montrer comment multiplier deux polynômes linéaires $ax + b$ et $cx + d$ à l'aide de trois multiplications seulement. (*conseil* : L'une des multiplications est $(a + b) \cdot (c + d)$.)
- Donner deux algorithmes diviser-pour-régner permettant de multiplier deux polynômes de degré borné par n en temps $\Theta(n^{\lg 3})$. Le premier algorithme devra diviser les coefficients des polynômes en deux moitiés, l'une supérieure et l'autre inférieure ; le second algorithme devra séparer les coefficients selon la parité des indices.

- c. Montrer que deux entiers à n bits peuvent être multipliés en $O(n^{\lg 3})$ étapes, où chaque étape s'exécute sur au plus un nombre constant de valeurs à 1 bit.

30.2. Matrices de Toeplitz

Une *matrice de Toeplitz* est une matrice $n \times n$ $A = (a_{ij})$ telle que $a_{ij} = a_{i-1, j-1}$ pour $i = 2, 3, \dots, n$ et $j = 2, 3, \dots, n$.

- a. La somme de deux matrices de Toeplitz est-elle nécessairement une matrice de Toeplitz ? Et le produit ?
- b. Décrire une manière de représenter une matrice de Toeplitz de sorte que deux matrices de Toeplitz $n \times n$ puissent être additionnées en temps $O(n)$.
- c. Donner un algorithme à temps $O(n \lg n)$ pour la multiplication d'une matrice $n \times n$ de Toeplitz par un vecteur de longueur n . Utiliser votre représentation de la partie (b).
- d. Donner un algorithme efficace permettant de multiplier deux matrices $n \times n$ de Toeplitz. Analyser son temps d'exécution.

30.3. Transformée rapide de Fourier multidimensionnelle

On peut généraliser la transformée discrète de Fourier monodimensionnelle, définie par l'équation (30.8), à d dimensions. L'entrée est un tableau à d -dimensions $A = (a_{i_1, i_2, \dots, i_d})$ dont les dimensions sont n_1, n_2, \dots, n_d , avec $n_1 n_2 \dots n_d = n$. On définit la transformée discrète de Fourier à d dimensions par l'équation

$$y_{k_1, k_2, \dots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \dots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \dots, j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \dots \omega_{n_d}^{j_d k_d}$$

pour $0 \leq k_1 < n_1, 0 \leq k_2 < n_2, \dots, 0 \leq k_d < n_d$.

- a. Montrer que l'on peut calculer une DFT à d dimensions en calculant des DFT monodimensionnelles sur chaque dimension prise à tour de rôle. Plus précisément, on commence par calculer n/n_1 DFT monodimensionnelles distinctes pour la dimension 1. Ensuite, en utilisant comme entrée le résultat de la DFT pour la dimension 1, on calcule n/n_2 DFT monodimensionnelles distinctes pour la dimension 2. En prenant ce résultat comme entrée, on calcule n/n_3 DFT monodimensionnelles distinctes pour la dimension 3, etc. jusqu'à la dimension d .
- b. Montrer que l'ordre des dimensions ne compte pas, de sorte que l'on peut calculer une DFT à d dimensions en calculant les DFT monodimensionnelles dans n'importe quel ordre des d dimensions.
- c. Montrer que, si l'on calcule chaque DFT monodimensionnelle en calculant la transformée rapide de Fourier, le temps total mis pour calculer une DFT à d dimensions est $O(n \lg n)$, indépendamment de d .

30.4. Évaluation de toutes les dérivées d'un polynôme en un point

Étant donné un polynôme $A(x)$ de degré borné par n , sa dérivée t ème est définie par

$$A^{(t)}(x) = \begin{cases} A(x) & \text{si } t = 0, \\ \frac{d}{dx}A^{(t-1)}(x) & \text{si } 1 \leq t \leq n-1, \\ 0 & \text{si } t \geq n. \end{cases}$$

A partir de la représentation par coefficients $(a_0, a_1, \dots, a_{n-1})$ de $A(x)$ et à partir d'un point donné x_0 , on souhaite déterminer $A^{(t)}(x_0)$ pour $t = 0, 1, \dots, n-1$.

a. Connaissant des coefficients b_0, b_1, \dots, b_{n-1} tels que

$$A(x) = \sum_{j=0}^{n-1} b_j(x-x_0)^j,$$

montrer comment calculer $A^{(t)}(x_0)$, pour $t = 0, 1, \dots, n-1$, en temps $O(n)$.

b. Expliquer comment trouver b_0, b_1, \dots, b_{n-1} en temps $O(n \lg n)$, connaissant $A(x_0 + \omega_n^k)$ pour $k = 0, 1, \dots, n-1$.

c. Démontrer que

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j)g(r-j) \right),$$

où $f(j) = a_j \cdot j!$ et

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{si } -(n-1) \leq l \leq 0, \\ 0 & \text{si } 1 \leq l \leq (n-1). \end{cases}$$

d. Expliquer comment évaluer $A(x_0 + \omega_n^k)$ pour $k = 0, 1, \dots, n-1$ en temps $O(n \lg n)$.
En conclure que toutes les dérivées non triviales de $A(x)$ peuvent être évaluées en x_0 en temps $O(n \lg n)$.

30.5. Évaluation polynomiale en des points multiples

Nous avons observé que le problème consistant à évaluer un polynôme de degré majoré par $n-1$ en un point unique pouvait être résolu en temps $O(n)$, à l'aide de la règle de Horner. Nous avons découvert également qu'un tel polynôme pouvait être évalué sur toutes les n racines complexes de l'unité en temps $O(n \lg n)$ à l'aide de la FFT. Nous allons à présent montrer comment évaluer en n points arbitraires, et en temps $O(n \lg^2 n)$, un polynôme de degré majoré par n .

Pour cela, on utilisera le fait qu'il est possible de calculer en temps $O(n \lg n)$ le reste du polynôme quand un tel polynôme est divisé par un autre, résultat que nous admettrons sans le démontrer. Par exemple, le reste de la division de $3x^3 + x^2 - 3x + 1$ par $x^2 + x + 2$ est

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5.$$

Étant donnée la représentation par coefficients d'un polynôme $A(x) = \sum_{k=0}^{n-1} a_k x^k$ et n points x_0, x_1, \dots, x_{n-1} , on désire calculer les n valeurs $A(x_0), A(x_1), \dots, A(x_{n-1})$. Pour $0 \leq i \leq j \leq n-1$, on définit les polynômes $P_{ij}(x) = \prod_{k=i}^j (x - x_k)$ et $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. Notez que $Q_{ij}(x)$ a un degré au plus égal à $j - i$.

- Démontrer que $A(x) \bmod (x - z) = A(z)$ pour un point z quelconque.
- Démontrer que $Q_{kk}(x) = A(x_k)$ et que $Q_{0,n-1}(x) = A(x)$.
- Démontrer que, pour $i \leq k \leq j$, on a $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ et $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.
- Donner un algorithme à temps $O(n \lg^2 n)$ pour évaluer $A(x_0), A(x_1), \dots, A(x_{n-1})$.

30.6. FFT et arithmétique modulaire

La manière dont nous avons défini la transformation discrète de Fourier impose l'utilisation des nombres complexes, ce qui peut conduire à une perte de précision due aux erreurs d'arrondis. Pour certains problèmes, on sait à l'avance que la réponse ne contiendra que des entiers et il est alors souhaitable d'utiliser une variante de la FFT basée sur l'arithmétique des congruences, de façon à être sûr que la réponse est calculée exactement. Un exemple de ce type de problème serait, par exemple, de multiplier deux polynômes à coefficients entiers. Une approche est donné à l'exercice 30.2.6, faisant appel à un modulo de longueur $\Omega(n)$ bits pour gérer une transformée discrète de Fourier sur n points. Le problème que voici propose une autre approche qui utilise un modulo de longueur plus raisonnable $O(\lg n)$; il impose que soient comprises les notions du chapitre 31.

Soit n une puissance de 2.

- On suppose que l'on recherche le plus petit k tel que $p = kn + 1$ est premier. Donner une démonstration heuristique simple qui justifie le fait que l'on peut s'attendre à ce que k vaille environ $\lg n$. (La valeur de k peut être beaucoup plus grande ou beaucoup plus petite, mais on peut raisonnablement espérer examiner $O(\lg n)$ valeurs candidates de k en moyenne.) Comparer la longueur attendue de p à la longueur de n .
Soit g un générateur de \mathbf{Z}_p^* et soit $w = g^k \bmod p$.
- Dire pourquoi la transformée discrète de Fourier et son inverse sont des opérations inverses bien définies modulo p , w étant utilisée comme racine n ème principale de l'unité.
- Prouver que la FFT et son inverse peuvent fonctionner modulo p en temps $O(n \lg n)$, en supposant que les opérations sur des mots de $O(\lg n)$ bits prennent un temps unitaire. On suppose que l'algorithme a comme paramètres p et w .
- Calculer la transformée discrète de Fourier modulo $p = 17$ du vecteur $(0, 5, 3, 7, 7, 2, 1, 6)$. Notez que $g = 3$ est un générateur de \mathbf{Z}_{17}^* .

NOTES

Le livre de VanLoan [303] est une référence en matière de présentation de la transformation rapide de Fourier. Press, Flannery, Teukolsky et Vetterling [248, 249] contient une bonne description de la transformation rapide de Fourier et de ses applications. Pour une excellente introduction au traitement du signal, domaine d'application célèbre de la FFT, voir les livres de Oppenheim et Schafer [232] et de Oppenheim et Willsky [233]. Le manuel de Oppenheim et Schafer montre aussi comment traiter les cas où n n'est pas une puissance entière de 2.

L'analyse de Fourier ne se limite pas aux données à 1 dimension. Elle sert énormément, dans le domaine du traitement d'image, pour analyser des données à 2 ou plusieurs dimensions. Les livres de Gonzalez et Woods [127] et Pratt [246] traitent des transformations de Fourier multidimensionnelles et de leurs applications au traitement d'image, alors que les manuels de Tolimieri, An et Lu [300] et Van Loan [303] traitent des mathématiques des transformations rapides de Fourier multidimensionnelles.

On attribue le plus souvent à Cooley et à Tukey [68] l'élaboration de la FFT dans les années 1960. La FFT a, en réalité, été découverte de nombreuses fois auparavant, mais son importance n'a été pleinement comprise qu'avec l'avènement des ordinateurs. Press, Flannery, Teukolsky et Vetterling attribuent les origines de la méthode à Runge et à König en 1924, mais un article de Heideman, Johnson et Burrus [141] fait remonter l'histoire de la FFT jusqu'à C. F. Gauss en 1805.

Chapitre 31

Algorithmes de la théorie des nombres

La théorie des nombres a été longtemps considérée comme un domaine des mathématiques pures tout à la fois fascinant et sans grande utilité. Aujourd'hui, les algorithmes de la théorie des nombres sont largement utilisés, notamment en raison de la création de modèles cryptographiques fondés sur les grands nombres premiers. La faisabilité de ces modèles dépend de notre capacité à trouver facilement de grands nombres premiers, alors que leur sécurité dépend de notre incapacité à factoriser le produit de grands nombres premiers. Ce chapitre présente quelques notions et algorithmes de la théorie des nombres, que l'on retrouve dans ce type d'applications.

La section 31.1 introduit les concepts de base de la théorie des nombres, comme la divisibilité, l'équivalence modulo et la factorisation unique. La section 31.2 étudie l'un des plus anciens algorithmes du monde : l'algorithme d'Euclide pour le calcul du plus grand commun diviseur de deux entiers. La section 31.3 passe en revue les concepts de l'arithmétique modulo. La section 31.4 étudie ensuite l'ensemble des multiples d'un nombre a donné, modulo n , et montre comment trouver toutes les solutions de l'équation $ax \equiv b \pmod{n}$ à l'aide de l'algorithme d'Euclide. Le théorème du reste chinois est présenté à la section 31.5. La section 31.6 considère les puissances d'un nombre a donné, modulo n , et présente un algorithme d'élevations répétées au carré permettant de calculer efficacement $a^b \pmod{n}$, connaissant a , b et n . Cette opération est au cœur de la vérification efficace du caractère premier d'un nombre et d'une bonne partie de la cryptographie moderne. La section 31.7 décrit

ensuite le système RSA de cryptographie à clé publique. La section 31.8 décrit un test de primalité randomisé qui permet de trouver efficacement de grands nombres premiers, tâche essentielle lors de la création de clés pour le cryptosystème RSA. Enfin, la section 31.9 permet de revoir une heuristique simple mais efficace de factorisation des entiers de petite taille. Il est amusant de constater que la factorisation est un problème dont on espère qu'il est intraitable. En effet, la sécurité de RSA repose sur la difficulté de factoriser les grands nombres entiers.

a) Taille des entrées et coût des calculs arithmétiques

Comme nous travaillerons sur des grands entiers, il faut adapter notre façon de considérer la taille d'une entrée et le coût des opérations arithmétiques élémentaires.

Dans ce chapitre, une « grande entrée » sera en général une entrée contenant de « grands entiers », plutôt qu'une entrée contenant « beaucoup d'entiers » (comme pour le tri). Donc, nous mesurerons la taille d'une entrée en *nombres de bits* requis pour la représenter, et non plus par seulement le nombre d'entiers qu'elle contient. Un algorithme ayant pour entrées entières a_1, a_2, \dots, a_k est un **algorithme en temps polynomial** s'il s'exécute dans un temps polynomial en $\lg a_1, \lg a_2, \dots, \lg a_k$, c'est-à-dire polynomial par rapport aux longueurs des représentations binaires de ses entrées.

Dans la plus grande partie de ce livre, nous avons trouvé qu'il était pratique de voir les opérations arithmétiques élémentaires (multiplication, division ou calcul de reste) comme des opérations primitives ne consommant qu'une seule unité de temps. En comptant le nombre d'opérations arithmétiques élémentaires effectuées par un algorithme, nous avons une base pour faire une estimation raisonnable du temps d'exécution réel de l'algorithme sur un ordinateur. Les opérations élémentaires peuvent cependant être gourmandes en temps de calcul, lorsque leurs entrées sont grandes. Il devient alors commode de mesurer le nombre d'**opérations binaire** requises par un algorithme de la théorie des nombres. Dans ce modèle, la multiplication de deux entiers de β bits par la méthode ordinaire utilise $\Theta(\beta^2)$ opérations binaires. De même, la division d'un entier de β bits par un entier plus petit, ou le calcul du reste de la division d'un entier de β bits par un entier plus petit, peut s'effectuer en temps $\Theta(\beta^2)$ par des algorithmes simples. (Voir exercice 31.1.11.) On connaît des méthodes plus rapides. Par exemple, une méthode diviser-pour-régner simple pour multiplier deux entiers de β bits a un temps d'exécution $\Theta(\beta^{\lg_2 3})$, et la méthode la plus rapide connue à ce jour a pour temps d'exécution $\Theta(\beta \lg \beta \lg \lg \beta)$. Toutefois, pour des raisons pratiques, l'algorithme en $\Theta(\beta^2)$ est souvent le meilleur, et nous utiliserons cette borne comme base de nos analyses.

Dans ce chapitre, les algorithmes sont généralement analysés à la fois en fonction du nombre d'opérations arithmétiques et en fonction du nombre d'opérations binaires qu'ils nécessitent.

31.1 NOTIONS DE THÉORIE DES NOMBRES

Cette section se propose de revoir brièvement certaines notions de théorie élémentaire des nombres concernant l'ensemble $\mathbf{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ des entiers et l'ensemble $\mathbf{N} = \{0, 1, 2, \dots\}$ des entiers naturels.

b) Divisibilité et diviseurs

La divisibilité d'un entier par un autre est l'une des notions centrales de la théorie des nombres. La notation $d \mid a$ (lire « d **divise** a ») signifie que $a = kd$ pour un certain entier k . Tout entier divise 0. Si $a > 0$ et $d \mid a$, alors $|d| \leq |a|$. Si $d \mid a$, alors on dit aussi que a est un **multiple** de d . Si d ne divise pas a , on écrit $d \nmid a$.

Si $d \mid a$ et $d \geq 0$, on dit que d est un **diviseur** de a . Notons que $d \mid a$ si et seulement si $-d \mid a$, de sorte qu'on ne perd pas de généralité en définissant les diviseurs comme étant positifs ou nuls, sachant que l'opposé d'un diviseur de a divise également a . Un diviseur d'un entier a non nul vaut au moins 1, mais n'est pas supérieur à $|a|$. Par exemple, les diviseurs de 24 sont 1, 2, 3, 4, 6, 8, 12 et 24.

Tout entier a est divisible par les **diviseurs triviaux** 1 et a . Les diviseurs non triviaux de a sont également appelés **facteurs** de a . Par exemple, les facteurs de 20 sont 2, 4, 5 et 10.

c) Nombres premiers et nombres composés

On dit d'un entier $a > 1$ dont les seuls diviseurs sont les diviseurs triviaux 1 et a que c'est un **nombre premier**. Les nombres premiers ont de nombreuses propriétés spécifiques et jouent un rôle fondamental en théorie des nombres. Les 20 premiers nombres premiers sont, dans l'ordre croissant,

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71

L'exercice 31.1.1 vous demande de démontrer qu'il existe une infinité de nombres premiers. On dit d'un entier $a > 1$ qui n'est pas premier qu'il est **composé**. Par exemple, 39 est composé car $3 \mid 39$. L'entier 1 est appelé **unité**, et il n'est ni premier ni composé. De même, l'entier 0 et tous les entiers négatifs ne sont ni premiers ni composés.

d) Théorème de la division, restes et équivalences modulo

Étant donné un entier n , on peut séparer les entiers qui sont multiples de n de ceux qui ne le sont pas. Une grande partie de la théorie des nombres est basée sur un affinement de ce partitionnement, obtenu en classant les non multiples de n selon la valeur du reste de leur division par n . Le théorème suivant est à la base de cet affinement. La démonstration de ce théorème ne sera pas donnée ici (voir notamment Niven et Zuckerman [231]).

Théorème 31.1 (Théorème de la division) *Pour tout entier a et tout entier positif n , il existe deux entiers q et r uniques tels que $0 \leq r < n$ et $a = qn + r$.*

La valeur $q = \lfloor a/n \rfloor$ est le **quotient** de la division. La valeur $r = a \bmod n$ est le **reste** de la division. On a $n \mid a$ si et seulement si $a \bmod n = 0$.

Les entiers peuvent être divisés en n classes d'équivalence en fonction du reste de leur division par n . La **classe d'équivalence modulo n** qui contient un entier a est

$$[a]_n = \{a + kn : k \in \mathbf{Z}\} .$$

Par exemple, $[3]_7 = \{\dots, -11, -4, 3, 10, 17, \dots\}$; cet ensemble se note aussi $[-4]_7$ et $[10]_7$. En utilisant la notation définie à la page 49, on peut dire que écrire $a \in [b]_n$ est la même chose qu'écrire $a \equiv b \pmod{n}$. L'ensemble de toutes ces classes d'équivalence est

$$\mathbf{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\} . \quad (31.1)$$

On voit souvent la définition

$$\mathbf{Z}_n = \{0, 1, \dots, n - 1\} , \quad (31.2)$$

qu'on devra comprendre comme étant équivalente à l'équation (31.1) en gardant à l'esprit que 0 représente $[0]_n$, 1 représente $[1]_n$, etc. ; chaque classe est représentée par son plus petit élément positif ou nul. Toutefois, il ne faut pas oublier les classes d'équivalences correspondantes. Par exemple, une référence à -1 comme membre de \mathbf{Z}_n est une référence à $[n - 1]_n$, puisque $-1 \equiv n - 1 \pmod{n}$.

e) *Diviseurs communs et plus grand communs diviseurs*

Si d est à la fois diviseur de a et de b , alors d est un **diviseur commun** de a et b . Par exemple, les diviseurs de 30 sont 1, 2, 3, 5, 6, 10, 15 et 30, et les diviseurs communs de 24 et 30 sont donc 1, 2, 3 et 6. Notez que 1 est un diviseur commun de deux entiers quelconques.

Les diviseurs communs vérifient une propriété importante :

$$d \mid a \text{ et } d \mid b \text{ implique } d \mid (a + b) \text{ et } d \mid (a - b) . \quad (31.3)$$

Plus généralement, on a

$$d \mid a \text{ et } d \mid b \text{ implique } d \mid (ax + by) \quad (31.4)$$

pour deux entiers x et y quelconques. De même, si $a \mid b$, alors on a soit $|a| \leq |b|$, soit $b = 0$, ce qui implique que

$$a \mid b \text{ et } b \mid a \text{ implique } a = \pm b . \quad (31.5)$$

Le **plus grand commun diviseur** de deux entiers a et b , non nuls en même temps, est le plus grand des diviseurs communs de a et b ; on l'écrit $\text{pgcd}(a, b)$. Par exemple, $\text{pgcd}(24, 30) = 6$, $\text{pgcd}(5, 7) = 1$, et $\text{pgcd}(0, 9) = 9$. Si a et b ne sont pas tous les deux nuls, alors $\text{pgcd}(a, b)$ est un entier compris entre 1 et $\min(|a|, |b|)$. On

convient que $\text{pgcd}(0, 0)$ est égal à 0 ; cette définition est nécessaire pour rendre les propriétés classiques de la fonction pgcd (comme l'équation (31.9) ci-après) valides universellement.

La fonction pgcd satisfait aux propriétés élémentaires suivantes :

$$\text{pgcd}(a, b) = \text{pgcd}(b, a), \quad (31.6)$$

$$\text{pgcd}(a, b) = \text{pgcd}(-a, b), \quad (31.7)$$

$$\text{pgcd}(a, b) = \text{pgcd}(|a|, |b|), \quad (31.8)$$

$$\text{pgcd}(a, 0) = |a|, \quad (31.9)$$

$$\text{pgcd}(a, ka) = |a| \quad \text{pour tout } k \in \mathbf{Z}. \quad (31.10)$$

Le théorème suivant donne une autre caractérisation, très utile, du pgcd .

Théorème 31.2 *Si a et b sont deux entiers quelconques, non nuls en même temps, alors $\text{pgcd}(a, b)$ est le plus petit élément positif de l'ensemble $\{ax + by : x, y \in \mathbf{Z}\}$ des combinaisons linéaires de a et b .*

Démonstration : Soit s la plus petite positive de ces combinaisons linéaires de a et b , et soit $s = ax + by$ pour un certain $x, y \in \mathbf{Z}$. Soit $q = \lfloor a/s \rfloor$. L'équation (3.8) implique alors

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy), \end{aligned}$$

et donc $a \bmod s$ est aussi une combinaison linéaire de a et b . Mais, puisque $0 \leq a \bmod s < s$, on a $a \bmod s = 0$, car s est la plus petite combinaison linéaire positive de ce type. Ainsi, $s \mid a$ et par un raisonnement identique, $s \mid b$. Donc, s est un diviseur commun de a et b , et ainsi $\text{pgcd}(a, b) \geq s$. L'équation (31.4) implique que $\text{pgcd}(a, b) \mid s$, puisque $\text{pgcd}(a, b)$ divise à la fois a et b et que s est une combinaison linéaire de a et b . Mais $\text{pgcd}(a, b) \mid s$ et $s > 0$ implique que $\text{pgcd}(a, b) \leq s$. En rapprochant $\text{pgcd}(a, b) \geq s$ et $\text{pgcd}(a, b) \leq s$, on obtient $\text{pgcd}(a, b) = s$; on en conclut que s est le plus grand commun diviseur de a et b . \square

Corollaire 31.3 *Pour deux entiers a et b quelconques, si $d \mid a$ et $d \mid b$ alors $d \mid \text{pgcd}(a, b)$.*

Démonstration : Ce corollaire se déduit de l'équation (31.4), puisque $\text{pgcd}(a, b)$ est une combinaison linéaire de a et b d'après le théorème 31.2. \square

Corollaire 31.4 *Pour deux entiers a et b quelconques, et pour tout entier n positif,*

$$\text{pgcd}(an, bn) = n \text{pgcd}(a, b).$$

Démonstration : Si $n = 0$, le corollaire est trivial. Si $n > 0$, alors $\text{pgcd}(an, bn)$ est le plus petit élément positif de l'ensemble $\{anx + bny\}$, qui vaut n fois le plus petit élément positif de l'ensemble $\{ax + by\}$. \square

Corollaire 31.5 *Pour trois entiers positifs n , a et b quelconques, si $n \mid ab$ et $\text{pgcd}(a, n) = 1$, alors $n \mid b$.*

Démonstration : La démonstration est laissée en exercice (voir exercice 31.1.4). \square

f) Entiers premiers entre eux

On dit de deux entiers a et b qu'ils sont **premiers entre eux** si leur seul diviseur commun est 1, c'est-à-dire si $\text{pgcd}(a, b) = 1$. Par exemple, 8 et 15 sont premiers entre eux, puisque les diviseurs de 8 sont 1, 2, 4 et 8, tandis que ceux de 15 sont 1, 3, 5 et 15. Le théorème suivant établit que si deux entiers sont chacun premier avec un même entier p , alors leur produit est premier avec p .

Théorème 31.6 *Pour trois entiers quelconques a , b et p , si $\text{pgcd}(a, p) = 1$ et $\text{pgcd}(b, p) = 1$, alors $\text{pgcd}(ab, p) = 1$.*

Démonstration : Le théorème 31.2 nous dit qu'il existe des entiers x, y, x' et y' tels que

$$\begin{aligned} ax + py &= 1, \\ bx' + py' &= 1. \end{aligned}$$

En multipliant ces équations et en les réorganisant, on obtient

$$ab(xx') + p(ybx' + y'ax + pyy') = 1.$$

L'entier 1 étant donc une combinaison linéaire positive de ab et p , il suffit d'utiliser le théorème 31.2 pour compléter la démonstration. \square

On dit que les entiers n_1, n_2, \dots, n_k sont **premiers entre eux deux à deux** si, pour tout couple $i \neq j$, on a $\text{pgcd}(n_i, n_j) = 1$.

g) Unicité de la factorisation

Voici un fait élémentaire mais important concernant la divisibilité par des nombres premiers.

Théorème 31.7 *Quel que soit le nombre premier p et quels que soient les entiers a , b , si $p \mid ab$, alors $p \mid a$ ou $p \mid b$ (ou les deux).*

Démonstration : Raisonnons par l'absurde en supposant que $p \mid ab$, mais que $p \nmid a$ et $p \nmid b$. Dans ce cas, $\text{pgcd}(a, p) = 1$ et $\text{pgcd}(b, p) = 1$, puisque les seuls diviseurs de p sont 1 et p , et que par hypothèse p ne divise ni a ni b . Le théorème 31.6 implique alors que $\text{pgcd}(ab, p) = 1$, ce qui contredit l'hypothèse $p \mid ab$, puisque $p \mid ab$ implique $\text{pgcd}(ab, p) = p$. Cette contradiction complète la démonstration. \square

Une conséquence du théorème 31.7 est qu'un entier a une seule décomposition possible en facteurs premiers.

Théorème 31.8 (Unicité de la factorisation) *Un entier composé a peut s'écrire d'une seule façon comme un produit de la forme*

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

où les p_i sont premiers, $p_1 < p_2 < \cdots < p_r$, et les e_i sont des entiers positifs.

Démonstration : La démonstration est l'objet de l'exercice 31.1.10. □

Par exemple, le nombre 6000 peut être décomposé de façon unique en $2^4 \cdot 3 \cdot 5^3$.

Exercices

31.1.1 Démontrer qu'il existe une infinité de nombres premiers. (*Conseil :* Montrer qu'aucun des nombres premiers p_1, p_2, \dots, p_k ne divise $(p_1 p_2 \cdots p_k) + 1$.)

31.1.2 Démontrer que, si $a \mid b$ et $b \mid c$, alors $a \mid c$.

31.1.3 Démontrer que, si p est premier et $0 < k < p$, alors $\text{pgcd}(k, p) = 1$.

31.1.4 Démontrer le corollaire 31.5.

31.1.5 Démontrer que, si p est premier et $0 < k < p$, alors $p \mid \binom{p}{k}$. En conclure que, quels que soient les entiers a, b et le nombre premier p ,

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

31.1.6 Démontrer que, si a et b sont deux entiers positifs tels que $a \mid b$, alors

$$(x \bmod b) \bmod a = x \bmod a$$

pour tout x . Démontrer, avec les mêmes hypothèses, que

$$x \equiv y \pmod{b} \text{ implique } x \equiv y \pmod{a}$$

pour deux entiers x et y quelconques.

31.1.7 Pour tout entier $k > 0$, on dit qu'un entier n est une *puissance k ième* s'il existe un entier a tel que $a^k = n$. On dit que $n > 1$ est une *puissance non triviale* si c'est une puissance k ième pour un certain entier $k > 1$. Montrer comment déterminer dans un temps polynomial en β si un entier n de β bits est une puissance non triviale.

31.1.8 Démontrer les équations (31.6)–(31.10).

31.1.9 Montrer que l'opérateur pgcd est associatif. Autrement dit, démontrer que pour trois entiers a , b et c quelconques,

$$\text{pgcd}(a, \text{pgcd}(b, c)) = \text{pgcd}(\text{pgcd}(a, b), c) .$$

31.1.10 * Démontrer le théorème 31.8.

31.1.11 Donner des algorithmes efficaces pour la division d'un entier de β bits par un entier plus petit, et pour le calcul du reste de la division d'un entier de β bits par un entier plus petit. Ces algorithmes devront s'exécuter en temps $O(\beta^2)$.

31.1.12 Donner un algorithme efficace permettant de convertir un entier (binaire) de β bits en sa représentation décimale. Montrer que si la multiplication ou la division d'entiers dont la longueur est au plus β nécessite un temps $M(\beta)$, alors la conversion binaire vers décimal peut s'effectuer en temps $\Theta(M(\beta) \lg \beta)$. (*Conseil* : Utiliser une approche diviser-pour-régner, en obtenant les moitiés supérieure et inférieure du résultat par des récursivités séparées.)

31.2 PLUS GRAND COMMUN DIVISEUR

Dans cette section, nous allons utiliser l'algorithme d'Euclide pour calculer efficacement le plus grand commun diviseur de deux entiers. L'analyse du temps d'exécution fait apparaître une relation surprenante avec les nombres de Fibonacci, qui donnent l'entrée la plus défavorable pour l'algorithme d'Euclide.

On se restreint dans cette section aux entiers positifs ou nuls. Cette restriction se justifie par l'équation (31.8), qui établit que $\text{pgcd}(a, b) = \text{pgcd}(|a|, |b|)$.

En principe, il est possible de calculer $\text{pgcd}(a, b)$ pour des entiers a et b positifs à partir de la décomposition en facteurs premiers de a et b . En effet, si

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \quad (31.11)$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} , \quad (31.12)$$

en utilisant des exposants nuls pour que l'ensemble de nombres premiers p_1, p_2, \dots, p_r soit le même pour a et b , alors, on a (ce que vous demandera de montrer l'exercice 31.2.1)

$$\text{pgcd}(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} . \quad (31.13)$$

Comme nous le verrons à la section 31.9, les meilleurs algorithmes de factorisation connus à ce jour ne s'exécutent pas en temps polynomial. Cette approche pour calculer le plus grand commun diviseur semble donc peu indiquée pour engendrer un algorithme efficace.

L'algorithme d'Euclide calculant le plus grand commun diviseurs est basé sur le théorème suivant.

Théorème 31.9 (Théorème de récursivité pour le PGCD) *Quel que soit l'entier a positif ou nul, et quel que soit l'entier positif b ,*

$$\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b).$$

Démonstration : Nous allons montrer que $\text{pgcd}(a, b)$ et $\text{pgcd}(b, a \bmod b)$ se divisent l'un l'autre, et donc qu'ils doivent être égaux, d'après l'équation (31.5) (puisque'ils sont tous les deux positifs ou nuls).

On commence par montrer que $\text{pgcd}(a, b) \mid \text{pgcd}(b, a \bmod b)$. Si l'on note $d = \text{pgcd}(a, b)$, alors $d \mid a$ et $d \mid b$. D'après l'équation (3.8), $(a \bmod b) = a - qb$, où $q = \lfloor a/b \rfloor$. Comme $(a \bmod b)$ est donc une combinaison linéaire de a et b , l'équation (31.4) implique que $d \mid (a \bmod b)$. Donc, puisque $d \mid b$ et $d \mid (a \bmod b)$, le corollaire 31.3 implique que $d \mid \text{pgcd}(b, a \bmod b)$ ou, si l'on préfère, que

$$\text{pgcd}(a, b) \mid \text{pgcd}(b, a \bmod b). \quad (31.14)$$

Montrer que $\text{pgcd}(b, a \bmod b) \mid \text{pgcd}(a, b)$ se fait pratiquement de la même manière. Si on prend maintenant $d = \text{pgcd}(b, a \bmod b)$, alors $d \mid b$ et $d \mid (a \bmod b)$. Comme $a = qb + (a \bmod b)$, où $q = \lfloor a/b \rfloor$, a est une combinaison linéaire de b et $(a \bmod b)$. D'après l'équation (31.4), on conclut que $d \mid a$. Comme $d \mid b$ et $d \mid a$, on a $d \mid \text{pgcd}(a, b)$ d'après le corollaire 31.3, ou de manière équivalente

$$\text{pgcd}(b, a \bmod b) \mid \text{pgcd}(a, b). \quad (31.15)$$

En utilisant l'égalité (31.5) pour combiner les équations (31.14) et (31.15), on termine la démonstration. \square

a) Algorithme d'Euclide

L'algorithme de pgcd ci-dessous est décrit dans les *Eléments* d'Euclide (300 av. J.-C. environ), bien qu'il soit peut être antérieur. Il est écrit comme un programme récursif, basé directement sur le théorème 31.9. Les entrées a et b sont deux entiers positifs ou nuls quelconques.

EUCLIDE(a, b)

- 1 **si** $b = 0$
- 2 **alors retourner** a
- 3 **sinon retourner** EUCLIDE($b, a \bmod b$)

On peut suivre l'exécution de EUCLIDE en considérant le calcul de $\text{pgcd}(30, 21)$, par exemple :

$$\begin{aligned} \text{EUCLIDE}(30, 21) &= \text{EUCLIDE}(21, 9) \\ &= \text{EUCLIDE}(9, 3) \\ &= \text{EUCLIDE}(3, 0) \\ &= 3. \end{aligned}$$

Dans ce calcul, il y a trois appels récursifs de EUCLIDE.

La validation de EUCLIDE repose sur le théorème 31.9 et sur le fait que, si a est retourné en ligne 2, alors $b = 0$, de sorte que l'équation (31.9) implique que $\text{pgcd}(a, b) = \text{pgcd}(a, 0) = a$. L'algorithme ne peut pas s'appeler de manière récursive indéfiniment, puisque le deuxième argument décroît strictement à chaque appel récursif et qu'il est toujours positif. EUCLIDE se termine donc toujours avec la réponse correcte.

b) Temps d'exécution de l'algorithme d'Euclide

On analyse le temps d'exécution de EUCLIDE dans le pire des cas en fonction de la taille de a et b . On suppose, sans remettre en cause le caractère général de la démonstration, que $a > b \geq 0$. Cette hypothèse peut se justifier par la remarque suivante : si $b > a \geq 0$, alors EUCLIDE(a, b) effectue immédiatement l'appel récursif EUCLIDE(b, a). Autrement dit, si le premier argument est inférieur au second, EUCLIDE utilise un appel récursif pour intervertir ses deux arguments avant de continuer. De même, si $b = a > 0$, la procédure se termine après un appel récursif, puisque $a \bmod b = 0$.

Le temps d'exécution global de EUCLIDE est proportionnel au nombre d'appels récursifs qu'il effectue. Notre analyse utilise les nombres de Fibonacci F_k , définis par la récurrence (3.21).

Lemme 31.10 *Si $a > b \geq 1$ et si l'invocation de EUCLIDE(a, b) effectue $k \geq 1$ appels récursifs, alors $a \geq F_{k+2}$ et $b \geq F_{k+1}$.*

Démonstration : La démonstration se fait par récurrence sur k . Pour la base, soit $k = 1$. Alors, $b \geq 1 = F_2$, et comme $a > b$, on doit avoir $a \geq 2 = F_3$. Comme $b > (a \bmod b)$, dans chaque appel récursif le premier argument est strictement plus grand que le second ; l'hypothèse selon laquelle $a > b$ est donc valable pour chaque appel récursif.

Supposons pour les besoins de la récurrence que le lemme soit vérifié si $k - 1$ appels récursifs sont effectués ; il faut alors démontrer qu'il est vrai pour k appels récursifs. Puisque $k > 0$, on a $b > 0$ et EUCLIDE(a, b) appelle EUCLIDE($b, a \bmod b$) récursivement, qui effectue à son tour $k - 1$ appels récursifs. L'hypothèse de récurrence implique alors que $b \geq F_{k+1}$ (démontrant ainsi une partie du lemme), et $(a \bmod b) \geq F_k$. On a

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor a/b \rfloor b) \\ &\leq a, \end{aligned}$$

puisque $a > b > 0$ implique $\lfloor a/b \rfloor \geq 1$. Donc,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2}. \end{aligned}$$

□

Le théorème suivant est un corollaire immédiat de ce lemme.

Théorème 31.11 (Théorème de Lamé) *Pour tout entier $k \geq 1$, si $a > b \geq 1$ et $b < F_{k+1}$, alors l'invocation de $\text{EUCLIDE}(a, b)$ engendre moins de k appels récursifs.*

On peut montrer que le majorant du théorème 31.11 est le meilleur possible. Les nombres consécutifs de Fibonacci représentent l'entrée la plus défavorable pour EUCLIDE . Comme $\text{EUCLIDE}(F_3, F_2)$ fait exactement un appel récursif et comme, pour $k \geq 2$, on a $F_{k+1} \bmod F_k = F_{k-1}$, on a également

$$\begin{aligned} \text{pgcd}(F_{k+1}, F_k) &= \text{pgcd}(F_k, (F_{k+1} \bmod F_k)) \\ &= \text{pgcd}(F_k, F_{k-1}). \end{aligned}$$

Donc, $\text{EUCLIDE}(F_{k+1}, F_k)$ s'appelle récursivement *exactement* $k - 1$ fois, atteignant ainsi le majorant donné par le théorème 31.11.

Comme F_k vaut approximativement $\phi^k / \sqrt{5}$, où ϕ est le nombre d'or $(1 + \sqrt{5})/2$ défini par l'équation (3.22), le nombre d'appels récursifs dans EUCLIDE est $O(\lg b)$. (Voir exercice 31.2.5 pour une borne plus fine.) On en déduit que, si EUCLIDE est appliqué à deux nombres de β bits, il effectue $O(\beta)$ opérations arithmétiques et $O(\beta^3)$ opérations binaires (en supposant que la multiplication et la division de deux nombres de β bits exigent $O(\beta^2)$ opérations binaires). Le problème 31.9.6 vous demandera d'établir une borne $O(\beta^2)$ pour le nombre d'opérations binaires.

c) Forme étendue de l'algorithme d'Euclide

Nous réécrivons à présent l'algorithme d'Euclide pour calculer quelques informations supplémentaires utiles. En particulier, nous généralisons l'algorithme pour lui permettre de calculer les coefficients entiers x et y tels que

$$d = \text{pgcd}(a, b) = ax + by. \quad (31.16)$$

Notez que x et y peuvent être nuls ou négatifs. Ces coefficients nous serviront plus tard pour le calcul des inverses pour la multiplication modulo. La procédure EUCLIDE-ETENDU prend en entrée une paire d'entiers positifs ou nuls et retourne un triplet de la forme (d, x, y) qui vérifie l'équation (31.16).

$\text{EUCLIDE-ETENDU}(a, b)$

- 1 **si** $b = 0$
- 2 **alors retourner** $(a, 1, 0)$
- 3 $(d', x', y') \leftarrow \text{EUCLIDE-ETENDU}(b, a \bmod b)$
- 4 $(d, x, y) \leftarrow (d', y', x' - \lfloor a/b \rfloor y')$
- 5 **retourner** (d, x, y)

La figure 31.1 illustre l'exécution de EUCLIDE-ETENDU avec le calcul de $\text{pgcd}(99, 78)$.

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	—	3	1	0

Figure 31.1 Un exemple de l'action de EUCLIDE-ETENDU sur les entrées 99 et 78. Chaque ligne montre un niveau de la récursivité : les valeurs des entrées a et b , la valeur calculée $\lfloor a/b \rfloor$, et les valeurs d , x et y retournées. Le triplet (d, x, y) retourné devient le triplet (d', x', y') utilisé dans le calcul fait au niveau suivant de la récursivité. L'appel EUCLIDE-ETENDU(99, 78) retourne $(3, -11, 14)$, et donc $\text{pgcd}(99, 78) = 3$ et $\text{pgcd}(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

La procédure EUCLIDE-ETENDU est une variante de la procédure EUCLIDE. La ligne 1 est équivalente au test « $b = 0$ » de la ligne 1 de EUCLIDE. Si $b = 0$, alors EUCLIDE-ETENDU retourne non seulement $d = a$ à la ligne 2, mais aussi les coefficients $x = 1$ et $y = 0$ tels que $a = ax + by$. Si $b \neq 0$, EUCLIDE-ETENDU commence par calculer (d', x', y') tels que $d' = \text{pgcd}(b, a \bmod b)$ et

$$d' = bx' + (a \bmod b)y' . \quad (31.17)$$

Comme pour EUCLIDE, on a dans ce cas $d = \text{pgcd}(a, b) = d' = \text{pgcd}(b, a \bmod b)$. Pour obtenir x et y tels que $d = ax + by$, on commence par réécrire l'équation (31.17) à l'aide de l'équation $d = d'$ et de l'équation (3.8) :

$$\begin{aligned} d &= bx' + (a - \lfloor a/b \rfloor b)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') . \end{aligned}$$

Donc, le choix $x = y'$ et $y = x' - \lfloor a/b \rfloor y'$ satisfait à l'équation $d = ax + by$, ce qui prouve la validité de EUCLIDE-ETENDU.

Comme le nombre d'appels récursifs effectués dans EUCLIDE est égal au nombre d'appels récursifs effectué dans EUCLIDE-ETENDU, les temps d'exécution de EUCLIDE et EUCLIDE-ETENDU sont les mêmes, à un facteur constant près. Autrement dit, pour $a > b > 0$, le nombre d'appels récursifs est $O(\lg b)$.

Exercices

31.2.1 Démontrer que les équations (31.11) et (31.12) impliquent l'équation (31.13).

31.2.2 Calculer les valeurs (d, x, y) retournées par l'appel EUCLIDE-ETENDU(899, 493).

31.2.3 Démontrer que, pour trois entiers a , k et n quelconques, on a

$$\text{pgcd}(a, n) = \text{pgcd}(a + kn, n) .$$

31.2.4 Réécrire EUCLIDE sous une forme itérative qui n'utilise qu'une quantité de mémoire constante (c'est-à-dire, qui ne stocke qu'un nombre constant de valeurs entières).

31.2.5 Si $a > b \geq 0$, montrer que l'invocation $\text{EUCLIDE}(a, b)$ effectue au plus $1 + \log_\phi b$ appels récursifs. Améliorer cette borne pour l'amener à $1 + \log_\phi(b/\text{pgcd}(a, b))$.

31.2.6 Quel est le résultat de $\text{EUCLIDE-ETENDU}(F_{k+1}, F_k)$? Démontrez que votre réponse est correcte.

31.2.7 On définit la fonction pgcd pour plus de deux arguments par l'équation récursive $\text{pgcd}(a_0, a_1, \dots, a_n) = \text{pgcd}(a_0, \text{pgcd}(a_1, \dots, a_n))$. Montrer que pgcd retourne la même réponse, indépendamment de l'ordre dans lequel ses arguments sont spécifiés. Montrer aussi comment trouver des entiers x_0, x_1, \dots, x_n tels que $\text{pgcd}(a_0, a_1, \dots, a_n) = a_0x_0 + a_1x_1 + \dots + a_nx_n$. Montrer que le nombre de divisions effectuées par votre algorithme est $O(n + \lg(\max\{a_0, a_1, \dots, a_n\}))$.

31.2.8 On définit $\text{ppcm}(a_1, a_2, \dots, a_n)$ comme le *plus petit commun multiple* des entiers a_1, a_2, \dots, a_n , c'est-à-dire le plus petit entier positif ou nul qui soit un multiple de chaque a_i . Montrer comment calculer $\text{ppcm}(a_1, a_2, \dots, a_n)$ efficacement en utilisant l'opération pgcd (à deux arguments) comme sous-programme.

31.2.9 Démontrez que n_1, n_2, n_3 et n_4 sont premiers entre eux deux à deux si et seulement si $\text{pgcd}(n_1n_2, n_3n_4) = \text{pgcd}(n_1n_3, n_2n_4) = 1$. Plus généralement, montrer que n_1, n_2, \dots, n_k sont premiers entre eux deux à deux si et seulement si l'on peut trouver $\lceil \lg k \rceil$ paires de nombres premiers entre eux parmi les n_i .

31.3 ARITHMÉTIQUE MODULAIRE

De façon informelle, on peut voir l'arithmétique modulaire comme l'arithmétique habituelle sur les entiers à ceci près que, si l'on travaille modulo n , tout résultat x est remplacé par l'élément de $\{0, 1, \dots, n-1\}$ qui est congru à x modulo n (autrement dit, x est remplacé par $x \bmod n$). Ce modèle informel est suffisant si l'on s'en tient aux opérations d'addition, de soustraction et de multiplication. Un modèle plus formel pour l'arithmétique modulaire, que nous allons donner maintenant, se décrit plus aisément dans le cadre de la théorie des groupes.

a) Groupes finis

Un *groupe* (S, \oplus) est un ensemble S associé à une opération binaire \oplus définie sur S qui vérifie les propriétés suivantes.

- 1) **Composition interne (fermeture)** : Quels que soient a et $b \in S$, on a $a \oplus b \in S$.
- 2) **Élément neutre** : Il existe un élément $e \in S$, appelé *élément neutre* du groupe, tel que $e \oplus a = a \oplus e = a$ pour tout $a \in S$.

- 3) **Associativité** : Quels que soient $a, b, c \in S$, on a $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
- 4) **Symétrique** : Pour chaque $a \in S$, il existe un élément $b \in S$ unique, appelé *symétrique de a* , tel que $a \oplus b = b \oplus a = e$.

Considérons l'exemple du groupe bien connu $(\mathbf{Z}, +)$ des entiers \mathbf{Z} muni de l'addition : 0 est l'élément neutre, et le symétrique de a est $-a$. Si un groupe (S, \oplus) respecte en plus *la loi de commutativité* $a \oplus b = b \oplus a$ pour tout $a, b \in S$, on dit alors qu'il est *abélien*. Si un groupe (S, \oplus) est tel que $|S| < \infty$, on dit que c'est un *groupe fini*.

b) Groupes définis par l'addition et la multiplication modulaires

On peut former deux groupes abéliens finis à l'aide de l'addition et la multiplication modulo n , où n est un entier strictement positif. Ces groupes sont basés sur les classes d'équivalence des entiers modulo n , définies à la section 31.1.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

\cdot_{15}	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	11	4	13	14	7	1
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(a)

(b)

Figure 31.2 Deux groupes finis. Les classes d'équivalence sont représentées par leur élément distingué. (a) Le groupe $(\mathbf{Z}_6, +_6)$. (b) Le groupe $(\mathbf{Z}_{15}^*, \cdot_{15})$.

Pour définir un groupe sur \mathbf{Z}_n , nous avons besoin de nouvelles opérations, que nous obtiendrons en redéfinissant les opérations habituelles d'addition et de multiplication. L'addition et la multiplication sont faciles à définir pour \mathbf{Z}_n , car la classe d'équivalence de deux entiers détermine de manière unique la classe d'équivalence de leur somme ou de leur produit. Autrement dit, si $a \equiv a' \pmod{n}$ et $b \equiv b' \pmod{n}$, alors

$$\begin{aligned}
 a + b &\equiv a' + b' \pmod{n}, \\
 ab &\equiv a'b' \pmod{n}.
 \end{aligned}$$

On peut donc définir l'addition et la multiplication modulo n , notée $+_n$ et \cdot_n , de la manière suivante :

$$\begin{aligned}
 [a]_n +_n [b]_n &= [a + b]_n, \\
 [a]_n \cdot_n [b]_n &= [ab]_n.
 \end{aligned}
 \tag{31.18}$$

(La soustraction peut se définir de la même façon sur \mathbf{Z}_n par $[a]_n -_n [b]_n = [a - b]_n$, mais le cas de la division est plus compliqué, comme on le verra.) Tout cela justifie la pratique répandue et commode qui veut qu'on utilise le plus petit élément positif de chaque classe d'équivalence comme représentant, lors des calculs dans \mathbf{Z}_n . L'addition, la soustraction, et la multiplication sont effectuées comme d'habitude sur les représentants, mais chaque résultat x est remplacé par le représentant de sa classe (c'est-à-dire par $x \bmod n$).

En se servant de cette définition de l'addition modulo n , on définit le **groupe additif modulo n** par $(\mathbf{Z}_n, +_n)$. La taille du groupe additif modulo n est $|\mathbf{Z}_n| = n$. La figure 31.2(a) donne la table d'addition pour le groupe $(\mathbf{Z}_6, +_6)$.

Théorème 31.12 *Le système $(\mathbf{Z}_n, +_n)$ est un groupe abélien fini.*

Démonstration : L'équation (31.18) montre que $(\mathbf{Z}_n, +_n)$ est fermé. L'associativité et la commutativité de $+_n$ se déduisent des propriétés de $+$:

$$\begin{aligned} ([a]_n +_n [b]_n) +_n [c]_n &= [a + b]_n +_n [c]_n \\ &= [(a + b) + c]_n \\ &= [a + (b + c)]_n \\ &= [a]_n +_n [b + c]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n) , \end{aligned}$$

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n \\ &= [b + a]_n \\ &= [b]_n +_n [a]_n . \end{aligned}$$

L'élément neutre de $(\mathbf{Z}_n, +_n)$ est 0 (c'est-à-dire $[0]_n$). Le symétrique additif (l'opposé) d'un élément a (autrement dit $[a]_n$) est l'élément $-a$ (c'est-à-dire $[-a]_n$ ou $[n - a]_n$), puisque $[a]_n +_n [-a]_n = [a - a]_n = [0]_n$. \square

À l'aide de la définition de la multiplication modulo n , on définit le **groupe multiplicatif modulo n** par $(\mathbf{Z}_n^*, \cdot_n)$. Les éléments de ce groupe forment l'ensemble \mathbf{Z}_n^* des éléments de \mathbf{Z}_n qui sont premiers avec n :

$$\mathbf{Z}_n^* = \{[a]_n \in \mathbf{Z}_n : \text{pgcd}(a, n) = 1\} .$$

Pour comprendre pourquoi \mathbf{Z}_n^* est bien défini, notez que pour $0 \leq a < n$, on a $a \equiv (a + kn) \pmod{n}$ pour tout entier k . D'après l'exercice 31.2.3, $\text{pgcd}(a, n) = 1$ implique donc $\text{pgcd}(a + kn, n) = 1$ pour tout entier k . Comme $[a]_n = \{a + kn : k \in \mathbf{Z}\}$, l'ensemble \mathbf{Z}_n^* est bien défini. Un exemple de ce type de groupe est

$$\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\} ,$$

où l'opération de groupe est la multiplication modulo 15. (Ici, un élément $[a]_{15}$ est noté a ; par exemple, $[7]_{15}$ est noté 7.) La figure 31.2(b) montre le groupe $(\mathbf{Z}_{15}^*, \cdot_{15})$. Par exemple, $8 \cdot 11 \equiv 13 \pmod{15}$, si l'on se place dans \mathbf{Z}_{15}^* . L'élément neutre pour ce groupe est 1.

Théorème 31.13 *Le système $(\mathbf{Z}_n^*, \cdot_n)$ est un groupe abélien fini.*

Démonstration : Le théorème 31.6 implique que $(\mathbf{Z}_n^*, \cdot_n)$ est fermé. L'associativité et la commutativité peuvent être démontrées pour \cdot_n comme elles l'ont été pour $+_n$ au cours de la démonstration du théorème 31.12. L'élément neutre est $[1]_n$. Pour montrer l'existence de symétriques, notons a un élément de \mathbf{Z}_n^* et (d, x, y) le résultat de EUCLIDE-ETENDU(a, n). Alors $d = 1$, puisque $a \in \mathbf{Z}_n^*$, et

$$ax + ny = 1$$

ou, si l'on préfère,

$$ax \equiv 1 \pmod{n}.$$

Donc, $[x]_n$ est un symétrique multiplicatif (ou inverse) de $[a]_n$, modulo n . La preuve de l'unicité des inverses est reportée au corollaire 31.26. \square

Comme exemple de calcul d'inverses, supposons que $a = 5$ et $n = 11$. Alors EUCLIDE-ETENDU(a, n) retourne $(d, x, y) = (1, -2, 1)$, de sorte que $1 = 5 \cdot (-2) + 11 \cdot 1$. Donc, -2 (c'est-à-dire, $9 \pmod{11}$) est un inverse de 5 modulo 11 .

Pour travailler sur les groupes $(\mathbf{Z}_n, +_n)$ et $(\mathbf{Z}_n^*, \cdot_n)$ dans le reste de ce chapitre, nous désignerons les classes d'équivalences par leurs représentants et nous représenterons les opérations $+_n$ et \cdot_n avec les notations arithmétiques usuelles $+$ et \cdot (ou la juxtaposition) respectivement. Par ailleurs, les équivalences modulo n peuvent aussi être interprétées comme des équations dans \mathbf{Z}_n . Par exemple, les deux énoncés suivants sont équivalents :

$$\begin{aligned} ax &\equiv b \pmod{n}, \\ [a]_n \cdot_n [x]_n &= [b]_n. \end{aligned}$$

Pour plus de commodité, on écrira parfois simplement S au lieu de (S, \oplus) , pour un groupe, lorsque l'opération associée pourra se déduire du contexte. On pourra donc se référer aux groupes $(\mathbf{Z}_n, +_n)$ et $(\mathbf{Z}_n^*, \cdot_n)$ avec les notations \mathbf{Z}_n et \mathbf{Z}_n^* , respectivement.

L'inverse (multiplicatif) d'un élément a est noté $(a^{-1} \pmod{n})$. La division dans \mathbf{Z}_n^* est définie par l'équation $a/b \equiv ab^{-1} \pmod{n}$. Par exemple, dans \mathbf{Z}_{15}^* on a $7^{-1} \equiv 13 \pmod{15}$, puisque $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, de sorte que $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

La taille de \mathbf{Z}_n^* se note $\phi(n)$. Cette fonction, connue sous le nom de **fonction phi d'Euler**, satisfait à l'équation

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right), \quad (31.19)$$

où p parcourt l'ensemble des nombres premiers qui divisent n (y compris n lui-même, si n est premier). On ne démontrera pas cette formule ici. Intuitivement, on commence avec une liste des n restes $\{0, 1, \dots, n-1\}$ et ensuite pour chaque nombre premier

p qui divise n , on élimine tous les multiples de p de la liste. Par exemple, puisque les diviseurs premiers de 45 sont 3 et 5,

$$\begin{aligned}\phi(45) &= 45 \left(1 - \frac{1}{3}\right) \left(1 - \frac{1}{5}\right) \\ &= 45 \left(\frac{2}{3}\right) \left(\frac{4}{5}\right) \\ &= 24.\end{aligned}$$

Si p est premier, alors $\mathbf{Z}_p^* = \{1, 2, \dots, p-1\}$, et

$$\phi(p) = p - 1. \quad (31.20)$$

Si n est composé, alors $\phi(n) < n - 1$.

c) Sous-groupes

Si (S, \oplus) est un groupe, si $S' \subseteq S$, et si (S', \oplus) est aussi un groupe, on dit alors que (S', \oplus) est un **sous-groupe** de (S, \oplus) . Par exemple, les entiers pairs forment un sous-groupe des entiers pour l'addition. Le théorème suivant fournit un outil utile de reconnaissance des sous-groupes.

Théorème 31.14 (Un sous-ensemble non vide et fermé d'un groupe fini est un sous-groupe) *Si (S, \oplus) est un groupe fini et si S' est un sous-ensemble non vide de S tel que $a \oplus b \in S'$ pour tout $a, b \in S'$, alors (S', \oplus) est un sous-groupe de (S, \oplus) .*

Démonstration : La démonstration est laissée en exercice (voir exercice 31.3.2). \square

À titre d'exemple, l'ensemble $\{0, 2, 4, 6\}$ forme un sous-groupe de \mathbf{Z}_8 , puisqu'il est non vide et fermé pour l'opération $+$ (autrement dit, fermé pour $+_8$).

Le théorème suivant fournit une contrainte extrêmement utile sur la taille d'un sous-groupe (la démonstration est omise).

Théorème 31.15 (Théorème de Lagrange) *Si (S, \oplus) est un groupe fini et (S', \oplus) est un sous-groupe de (S, \oplus) , alors $|S'|$ est un diviseur de $|S|$.*

On dit du sous-groupe S' d'un groupe S qu'il est **propre** si $S' \neq S$. Le corollaire ci-après nous servira dans l'analyse du test de primalité de Miller-Rabin à la section 31.8.

Corollaire 31.16 *Si S' est un sous-groupe propre d'un groupe fini S , alors $|S'| \leq |S|/2$.*

d) Sous-groupes engendrés par un élément

Le théorème 31.14 donne un moyen intéressant d'engendrer un sous-groupe d'un groupe fini (S, \oplus) : on choisit un élément a et on prend tous les éléments qui peuvent

être générés à partir de a en utilisant l'opération de groupe. En particulier, définissons $a^{(k)}$ pour $k \geq 1$ par

$$a^{(k)} = \bigoplus_{i=1}^k a = \underbrace{a \oplus a \oplus \cdots \oplus a}_k .$$

Par exemple, si l'on prend $a = 2$ dans le groupe \mathbf{Z}_6 , la séquence $a^{(1)}, a^{(2)}, \dots$ vaudra

$$2, 4, 0, 2, 4, 0, 2, 4, 0, \dots .$$

Dans le groupe \mathbf{Z}_n , on a $a^{(k)} = ka \pmod n$, et dans le groupe \mathbf{Z}_n^* , on a $a^{(k)} = a^k \pmod n$. Le *sous-groupe engendré par a* , noté $\langle a \rangle$ ou $(\langle a \rangle, \oplus)$, est défini par

$$\langle a \rangle = \{a^{(k)} : k \geq 1\} .$$

On dit que a *engendre* le sous-groupe $\langle a \rangle$ ou que a est un *générateur* de $\langle a \rangle$. Comme S est fini, $\langle a \rangle$ est un sous-ensemble fini de S , pouvant inclure jusqu'à S tout entier. Comme l'associativité de \oplus implique

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)} ,$$

$\langle a \rangle$ est fermé et donc, d'après le théorème 31.14, $\langle a \rangle$ est un sous-groupe de S . Par exemple, dans \mathbf{Z}_6 , on a

$$\begin{aligned} \langle 0 \rangle &= \{0\} , \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\} , \\ \langle 2 \rangle &= \{0, 2, 4\} . \end{aligned}$$

De même, dans \mathbf{Z}_7^* , on a

$$\begin{aligned} \langle 1 \rangle &= \{1\} , \\ \langle 2 \rangle &= \{1, 2, 4\} , \\ \langle 3 \rangle &= \{1, 2, 3, 4, 5, 6\} . \end{aligned}$$

L'*ordre* de a (dans le groupe S), noté $\text{ord}(a)$, est le plus petit $t > 0$ tel que $a^{(t)} = e$.

Théorème 31.17 *Pour un groupe fini (S, \oplus) quelconque, et pour tout $a \in S$, l'ordre d'un élément est égal à la taille du sous-groupe qu'il engendre, c'est-à-dire $\text{ord}(a) = |\langle a \rangle|$.*

Démonstration : Soit $t = \text{ord}(a)$. Puisque $a^{(t)} = e$ et $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ pour $k \geq 1$, si $i > t$, alors $a^{(i)} = a^{(j)}$ pour un certain $j < i$. Donc, aucun nouvel élément n'apparaît après $a^{(t)}$, de sorte que $\langle a \rangle = \{a^{(1)}, a^{(2)}, \dots, a^{(t)}\}$ et $|\langle a \rangle| \leq t$. Pour montrer que $|\langle a \rangle| = t$, on suppose *a contrario* que $a^{(i)} = a^{(j)}$ pour un certain couple i, j satisfaisant $1 \leq i < j \leq t$. Dans ce cas, $a^{(i+k)} = a^{(j+k)}$ pour $k \geq 0$. Mais cela implique que $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, ce qui est une contradiction, puisque $i + (t - j) < t$ alors que t est la plus petite valeur positive telle que $a^{(t)} = e$. Donc, chaque élément de la séquence $a^{(1)}, a^{(2)}, \dots, a^{(t)}$ est distinct, et $|\langle a \rangle| \geq t$. On en conclut que $\text{ord}(a) = |\langle a \rangle|$. \square

Corollaire 31.18 La séquence $a^{(1)}, a^{(2)}, \dots$ est périodique de période $t = \text{ord}(a)$; autrement dit, $a^{(i)} = a^{(j)}$ si et seulement si $i \equiv j \pmod{t}$.

Il est cohérent avec le corollaire précédent de définir $a^{(0)}$ comme étant égal à e et $a^{(i)}$ comme étant égal à $a^{(i \bmod t)}$, avec $t = \text{ord}(a)$, pour tout entier i .

Corollaire 31.19 Si (S, \oplus) est un groupe fini ayant pour élément neutre e , alors pour tout $a \in S$,

$$a^{(|S|)} = e .$$

Démonstration : Le théorème de Lagrange implique $\text{ord}(a) \mid |S|$, et donc $|S| \equiv 0 \pmod{t}$, où $t = \text{ord}(a)$. Par conséquent, $a^{(|S|)} = a^{(0)} = e$. \square

Exercices

31.3.1 Dessiner les tables d'opération pour les groupes $(\mathbf{Z}_4, +_4)$ et $(\mathbf{Z}_5^*, \cdot_5)$. Montrer que ces groupes sont isomorphes en exhibant une bijection α entre leurs éléments telle que $a + b \equiv c \pmod{4}$ si et seulement si $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

31.3.2 Démontrer le théorème 31.14.

31.3.3 Montrer que, si p est premier et e un entier positif, alors

$$\phi(p^e) = p^{e-1}(p-1) .$$

31.3.4 Montrer que, pour tout $n > 1$ et pour tout $a \in \mathbf{Z}_n^*$, la fonction $f_a : \mathbf{Z}_n^* \rightarrow \mathbf{Z}_n^*$ définie par $f_a(x) = ax \bmod n$ est une permutation de \mathbf{Z}_n^* .

31.3.5 Énumérer tous les sous-groupes de \mathbf{Z}_9 et de \mathbf{Z}_{13}^* .

31.4 RÉOLUTION D'ÉQUATIONS LINÉAIRES MODULAIRES

On considère le problème consistant à trouver des solutions à l'équation

$$ax \equiv b \pmod{n} , \tag{31.21}$$

où $a > 0$ et $n > 0$. Il existe plusieurs applications pour ce problème ; entre autres, on l'utilisera dans une partie de la procédure de calcul de clé du cryptosystème à clé publique RSA traité à la section 31.7. On suppose que a , b et n sont donnés, et on doit trouver toutes les valeurs de x modulo n qui vérifient l'équation (31.21). Il peut y avoir zéro, une ou plusieurs solutions.

Soit $\langle a \rangle$ le sous-groupe de \mathbf{Z}_n engendré par a . Comme

$$\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\},$$

l'équation (31.21) a une solution si et seulement si $b \in \langle a \rangle$. Le théorème de Lagrange (théorème 31.15) nous dit que $|\langle a \rangle|$ doit être un diviseur de n . Le théorème suivant nous donne une caractérisation précise de $\langle a \rangle$.

Théorème 31.20 *Pour deux entiers quelconques strictement positifs a et n , si $d = \text{pgcd}(a, n)$, alors*

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \dots, ((n/d) - 1)d\}, \quad (31.22)$$

dans \mathbf{Z}_n , et donc

$$|\langle a \rangle| = n/d.$$

Démonstration : On commence par montrer que $d \in \langle a \rangle$. Rappelez-vous que EUCLIDE-ETENDU(a, n) fournit des entiers x' et y' tels que $ax' + ny' = d$. Donc, $ax' \equiv d \pmod{n}$, de sorte que $d \in \langle a \rangle$.

Comme $d \in \langle a \rangle$, il s'ensuit que tout multiple de d appartient à $\langle a \rangle$, car un multiple d'un multiple de a est encore un multiple de a . Donc, $\langle a \rangle$ contient tous les éléments de $\{0, d, 2d, \dots, ((n/d) - 1)d\}$. Autrement dit, $\langle d \rangle \subseteq \langle a \rangle$.

Montrons à présent que $\langle a \rangle \subseteq \langle d \rangle$. Si $m \in \langle a \rangle$, alors $m = ax \bmod n$ pour un certain entier x , et donc $m = ax + ny$ pour un certain entier y . Toutefois, $d \mid a$ et $d \mid n$, et donc $d \mid m$ d'après l'équation (31.4). On a donc $m \in \langle d \rangle$.

En combinant ces résultats, on obtient $\langle a \rangle = \langle d \rangle$. Pour voir que $|\langle a \rangle| = n/d$, remarquons qu'il existe exactement n/d multiples de d compris entre 0 et $n - 1$ inclus. \square

Corollaire 31.21 *L'équation $ax \equiv b \pmod{n}$ a une solution pour l'inconnue x si et seulement si $\text{pgcd}(a, n) \mid b$.*

Corollaire 31.22 *L'équation $ax \equiv b \pmod{n}$ possède soit d solutions distinctes modulo n , où $d = \text{pgcd}(a, n)$, soit aucune solution.*

Démonstration : Si $ax \equiv b \pmod{n}$ possède une solution, alors $b \in \langle a \rangle$. D'après le théorème 31.17, $\text{ord}(a) = |\langle a \rangle|$, et donc le corollaire 31.18 et le théorème 31.20 impliquent que la séquence $ai \bmod n$, pour $i = 0, 1, \dots$, est périodique de période $|\langle a \rangle| = n/d$. Si $b \in \langle a \rangle$, alors b apparaît exactement d fois dans la séquence $ai \bmod n$, pour $i = 0, 1, \dots, n - 1$, puisque le bloc de valeurs $\langle a \rangle$ de longueur (n/d) est répété exactement d fois quand i augmente de 0 à $n - 1$. Les indices x des d positions pour lesquelles $ax \bmod n = b$ sont les solutions de l'équation $ax \equiv b \pmod{n}$. \square

Théorème 31.23 *Soit $d = \text{pgcd}(a, n)$, et supposons que $d = ax' + ny'$ pour deux entiers x' et y' particuliers (calculés, par exemple, par EUCLIDE-ETENDU). Si $d \mid b$, l'une des solutions de l'équation $ax \equiv b \pmod{n}$ est x_0 , où*

$$x_0 = x'(b/d) \bmod n.$$

Démonstration : On a

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} \\ &\equiv b \pmod{n}, \end{aligned}$$

et x_0 est donc une solution de $ax \equiv b \pmod{n}$. \square

Théorème 31.24 *On suppose que l'équation $ax \equiv b \pmod{n}$ admet une solution (autrement dit, $d \mid b$, avec $d = \text{pgcd}(a, n)$) et que x_0 est une des solutions de cette équation. Alors, cette équation possède exactement d solutions distinctes, modulo n , données par $x_i = x_0 + i(n/d)$ pour $i = 1, 2, \dots, d - 1$.*

Démonstration : Comme $n/d > 0$ et $0 \leq i(n/d) < n$ pour $i = 0, 1, \dots, d - 1$, les valeurs x_0, x_1, \dots, x_{d-1} sont toutes distinctes modulo n . Comme x_0 est une solution de $ax \equiv b \pmod{n}$, on a $ax_0 \pmod{n} = b$. Donc, pour $i = 0, 1, \dots, d - 1$, on a

$$\begin{aligned} ax_i \pmod{n} &= a(x_0 + i(n/d)) \pmod{n} \\ &= (ax_0 + ain/d) \pmod{n} \\ &= ax_0 \pmod{n} \quad (d \mid a) \\ &= b, \end{aligned}$$

et donc x_i est une solution, aussi. D'après le corollaire 31.22 il y a exactement d solutions, de sorte que x_0, x_1, \dots, x_{d-1} sont justement ces solutions. \square

Nous avons à présent fait le tour des notions mathématiques nécessaires pour résoudre l'équation $ax \equiv b \pmod{n}$; l'algorithme suivant imprime toutes les solutions de cette équation. Les entrées a et n sont des entiers strictement positifs arbitraires, et b est un entier arbitraire.

RÉSOLUTION-EQUATIONS-LINÉAIRES-MODULAIRES(a, b, n)

- 1 $(d, x', y') \leftarrow \text{EUCLIDE-ETENDU}(a, n)$
- 2 **si** $d \mid b$
- 3 **alors** $x_0 \leftarrow x'(b/d) \pmod{n}$
- 4 **pour** $i \leftarrow 0$ à $d - 1$
- 5 **faire** imprimer $(x_0 + i(n/d)) \pmod{n}$
- 6 **sinon** imprimer « pas de solution »

Pour prendre un exemple d'exécution de cette procédure, considérons l'équation $14x \equiv 30 \pmod{100}$ (ici, $a = 14$, $b = 30$ et $n = 100$). En appelant EUCLIDE-ETENDU à la ligne 1, on obtient $(d, x, y) = (2, -7, 1)$. Comme $2 \mid 30$, les lignes 3–5 sont exécutées. À la ligne 3, on calcule $x_0 = (-7)(15) \pmod{100} = 95$. La boucle des lignes 4–5 imprime les deux solutions 95 et 45.

La procédure RÉSOLUTION-EQUATIONS-LINÉAIRES-MODULAIRES fonctionne de la manière suivante. La ligne 1 calcule $d = \text{pgcd}(a, n)$ ainsi que deux valeurs x' et y' telles que $d = ax' + ny'$, démontrant ainsi que x' est une solution de l'équation $ax' \equiv d \pmod{n}$. Si d ne divise pas b , alors l'équation $ax \equiv b \pmod{n}$ n'a pas

de solution, d'après le corollaire 31.21. La ligne 2 teste si $d \mid b$; si tel n'est pas le cas, la ligne 6 indique qu'aucune solution n'existe. Autrement, la ligne 3 calcule une solution x_0 de $ax \equiv b \pmod{n}$, conformément au théorème 31.23. Étant donnée une solution, le théorème 31.24 dit que les $d - 1$ autres solutions peuvent être obtenues en ajoutant des multiples de (n/d) modulo n . La boucle **pour** des lignes 4–5 imprime les d solutions, en commençant par x_0 , et espacées de (n/d) modulo n .

Le nombre d'opérations arithmétiques nécessaires à RÉSOLUTION-EQUATIONS-LINÉAIRES-MODULAIRES est $O(\lg n + \text{pgcd}(a, n))$, puisque $O(\lg n)$ opérations arithmétiques sont faites par EUCLIDE-ÉTENDU et que chaque itération de la boucle **pour** des lignes 4–5 fait un nombre constant d'opérations arithmétiques.

Voici des corollaires, du plus grand intérêt, du théorème 31.24.

Corollaire 31.25 *Pour tout $n > 1$, si $\text{pgcd}(a, n) = 1$, alors l'équation $ax \equiv b \pmod{n}$ possède une solution unique modulo n .*

Si $b = 1$, cas fréquent d'un intérêt considérable, l'inconnue x que nous cherchons est un **inverse multiplicatif** de a , modulo n .

Corollaire 31.26 *Pour tout $n > 1$, si $\text{pgcd}(a, n) = 1$, alors l'équation*

$$ax \equiv 1 \pmod{n}$$

possède une solution unique, modulo n . Sinon, elle n'a pas de solution.

Le corollaire 31.26 nous permet d'utiliser la notation $(a^{-1} \pmod{n})$ pour désigner l'unique inverse multiplicatif de a modulo n , quand a et n sont premiers entre eux. Si $\text{pgcd}(a, n) = 1$, alors une solution à l'équation $ax \equiv 1 \pmod{n}$ est l'entier x retourné par EUCLIDE-ÉTENDU, puisque l'équation

$$\text{pgcd}(a, n) = 1 = ax + ny$$

implique $ax \equiv 1 \pmod{n}$. Donc, $(a^{-1} \pmod{n})$ peut être calculé efficacement à l'aide de EUCLIDE-ÉTENDU.

Exercices

31.4.1 Trouver toutes les solutions à l'équation $35x \equiv 10 \pmod{50}$.

31.4.2 Démontrer que l'équation $ax \equiv ay \pmod{n}$ implique $x \equiv y \pmod{n}$ si $\text{pgcd}(a, n) = 1$. Montrer que la condition $\text{pgcd}(a, n) = 1$ est nécessaire en donnant un contre-exemple avec $\text{pgcd}(a, n) > 1$.

31.4.3 On considère la modification suivante, à la ligne 3 de RÉSOLUTION-EQUATIONS-LINÉAIRES-MODULAIRES :

$$3 \quad \text{alors } x_0 \leftarrow x'(b/d) \bmod (n/d)$$

L'algorithme est-il toujours valide ? Pourquoi ?

31.4.4 * Soit $f(x) \equiv f_0 + f_1x + \dots + f_t x^t \pmod{p}$ un polynôme de degré t , à coefficients f_i pris dans \mathbf{Z}_p , p étant premier. On dit que $a \in \mathbf{Z}_p$ est un **zéro** de f si $f(a) \equiv 0 \pmod{p}$. Démontrer que, si a est un zéro de f , alors $f(x) \equiv (x - a)g(x) \pmod{p}$ pour un certain polynôme $g(x)$ de degré $t - 1$. Démontrer par récurrence sur t qu'un polynôme $f(x)$ de degré t peut avoir au plus t zéros distincts, modulo un nombre premier p .

31.5 THÉORÈME DU RESTE CHINOIS

Autour de 100 AP. J.-C., le mathématicien chinois Sun-Tsü résolut le problème consistant à trouver les entiers x dont la division par 3, 5 et 7 donne les restes 2, 3 et 2 respectivement. L'une de ces solutions est $x = 23$; toutes les solutions sont de la forme $23 + 105k$ pour des entiers arbitraires k . Le « théorème du reste chinois » fait correspondre un système d'équations modulo un ensemble d'entiers premiers entre eux deux à deux (par exemple 3, 5 et 7) à une équation modulo leur produit (par exemple, 105).

Le théorème du reste chinois possède deux intérêts principaux. Soit l'entier $n = n_1 n_2 \dots n_k$, où les facteurs n_i sont premiers entre eux deux à deux. Primo, le théorème du reste chinois est un « théorème de structure » qui décrit la structure de \mathbf{Z}_n comme étant identique à celle du produit cartésien $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \dots \times \mathbf{Z}_{n_k}$ dans lequel on associe l'addition et la multiplication modulo n_i à la i ème composante du produit. Secundo, cette description peut souvent être employée pour définir des algorithmes efficaces, puisque le travail dans chacun des systèmes \mathbf{Z}_{n_i} peut être plus efficace (en termes d'opérations de bits) que le travail modulo n .

Théorème 31.27 (Théorème du reste chinois) *Soit $n = n_1 n_2 \dots n_k$, où les n_i sont premiers entre eux deux à deux. On considère la correspondance*

$$a \leftrightarrow (a_1, a_2, \dots, a_k), \quad (31.23)$$

avec $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$ et

$$a_i = a \bmod n_i$$

pour $i = 1, 2, \dots, k$. Alors, la correspondance (31.23) est une bijection entre \mathbf{Z}_n et le produit cartésien $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \dots \times \mathbf{Z}_{n_k}$. Les opérations sur les éléments de \mathbf{Z}_n peuvent être effectuées de manière équivalente sur les k -uplets correspondants via application indépendante sur chaque composante de coordonnée. Autrement dit, si

$$a \leftrightarrow (a_1, a_2, \dots, a_k),$$

$$b \leftrightarrow (b_1, b_2, \dots, b_k),$$

alors

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \dots, (a_k + b_k) \bmod n_k), \quad (31.24)$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \dots, (a_k - b_k) \bmod n_k), \quad (31.25)$$

$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \dots, a_k b_k \bmod n_k). \quad (31.26)$$

Démonstration : Le passage entre ces deux représentations est quasi immédiat. Passer de a à (a_1, a_2, \dots, a_k) est très facile et ne nécessite que k divisions. Calculer a à partir des entrées (a_1, a_2, \dots, a_k) est un peu plus compliqué, et voici la manœuvre. On commence par définir $m_i = n/n_i$ pour $i = 1, 2, \dots, k$; donc m_i est le produit de tous les n_j autres que n_i : $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. On définit ensuite

$$c_i = m_i(m_i^{-1} \bmod n_i) \quad (31.27)$$

pour $i = 1, 2, \dots, k$. L'équation (31.27) est toujours bien définie : comme m_i et n_i sont premiers entre eux (d'après le théorème 31.6), le corollaire 31.26 assure que $(m_i^{-1} \bmod n_i)$ existe. Enfin, on peut calculer a en tant que fonction de a_1, a_2, \dots, a_k de la façon suivante :

$$a \equiv (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \pmod{n}. \quad (31.28)$$

Montrons maintenant que l'équation (31.28) assure que $a \equiv a_i \pmod{n_i}$ pour $i = 1, 2, \dots, k$. Notez que, si $j \neq i$, alors $m_j \equiv 0 \pmod{n_i}$, ce qui implique que $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Notez aussi que $c_i \equiv 1 \pmod{n_i}$, d'après l'équation (31.27). On a donc la correspondance sympathique et utile

$$c_i \leftrightarrow (0, 0, \dots, 0, 1, 0, \dots, 0),$$

vecteur qui a des 0 partout sauf à la i ème coordonnée, où il a un 1 ; les c_i forment donc une « base » pour la représentation, en un certain sens. Pour tout i , on a donc

$$\begin{aligned} a &\equiv a_i c_i && \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i}, \end{aligned}$$

ce qui est ce que nous voulions montrer : notre méthode de calcul de a à partir des a_i produit un résultat a qui vérifie les contraintes $a \equiv a_i \pmod{n_i}$ pour $i = 1, 2, \dots, k$. La correspondance est bijective, car on peut convertir dans les deux sens. Enfin, les équations (31.24)–(31.26) découlent directement de l'exercice 31.1.6, car $x \bmod n_i = (x \bmod n) \bmod n_i$ pour tout x et $i = 1, 2, \dots, k$. \square

Les corollaires suivants seront utilisés plus tard dans ce chapitre.

Corollaire 31.28 *Si n_1, n_2, \dots, n_k sont premiers entre eux deux à deux et si $n = n_1 n_2 \cdots n_k$, alors quels que soient les entiers a_1, a_2, \dots, a_k , l'ensemble d'équations simultanées*

$$x \equiv a_i \pmod{n_i},$$

pour $i = 1, 2, \dots, k$, possède une solution unique modulo n pour l'inconnue x .

Corollaire 31.29 Si n_1, n_2, \dots, n_k sont premiers entre eux deux à deux et si $n = n_1 n_2 \cdots n_k$, alors pour deux entiers x et a quelconques,

$$x \equiv a \pmod{n_i}$$

pour $i = 1, 2, \dots, k$ si et seulement si

$$x \equiv a \pmod{n}.$$

À titre d'exemple d'application du théorème du reste chinois, donnons-nous les deux équations

$$a \equiv 2 \pmod{5},$$

$$a \equiv 3 \pmod{13},$$

de sorte que $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$ et $n_2 = m_1 = 13$. On désire calculer $a \pmod{65}$, puisque $n = 65$. Comme $13^{-1} \equiv 2 \pmod{5}$ et $5^{-1} \equiv 8 \pmod{13}$, on a

$$c_1 = 13(2 \pmod{5}) = 26,$$

$$c_2 = 5(8 \pmod{13}) = 40,$$

et

$$a \equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65}$$

$$\equiv 52 + 120 \pmod{65}$$

$$\equiv 42 \pmod{65}.$$

La figure 31.3 donne une illustration du théorème du reste chinois, modulo 65.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figure 31.3 Une illustration du théorème du reste chinois, pour $n_1 = 5$ et $n_2 = 13$. Pour cet exemple, $c_1 = 26$ et $c_2 = 40$. La valeur de a modulo 65 est montrée à l'intersection de la ligne i et de la colonne j , avec $(a \pmod{5}) = i$ et $(a \pmod{13}) = j$. Notez que l'intersection de la ligne 0 et de la colonne 0 contient la valeur 0. De même, l'intersection de la ligne 4 et de la colonne 12 contient 64 (équivalent à -1). Comme $c_1 = 26$, descendre d'une ligne vers le bas augmente a de 26. De même, $c_2 = 40$ signifie que se déplacer d'une colonne vers la droite augmente a de 40. L'augmentation de a de 1 correspond à un déplacement diagonal vers la droite et vers le bas, sachant que, arrivé à la fin du tableau, on repart du haut de la colonne suivante ou de la droite.

On peut donc travailler modulo n , soit directement, soit dans la représentation transformée à l'aide de calcul séparés modulo n_i . Les calculs sont rigoureusement équivalents.

Exercices

31.5.1 Trouver toutes les solutions aux équations $x \equiv 4 \pmod{5}$ et $x \equiv 5 \pmod{11}$.

31.5.2 Trouver tous les entiers x dont la division par 9, 8 et 7 donne les restes 1, 2 et 3 respectivement.

31.5.3 Montrer que, avec les définitions du théorème 31.27, si $\text{pgcd}(a, n) = 1$, alors

$$(a^{-1} \pmod{n}) \leftrightarrow ((a_1^{-1} \pmod{n_1}), (a_2^{-1} \pmod{n_2}), \dots, (a_k^{-1} \pmod{n_k})) .$$

31.5.4 Avec les définitions du théorème 31.27, démontrer que, pour tout polynôme f , le nombre de racines de l'équation $f(x) \equiv 0 \pmod{n}$ est égal au produit du nombre de racines de chacune des équations $f(x) \equiv 0 \pmod{n_1}, f(x) \equiv 0 \pmod{n_2}, \dots, f(x) \equiv 0 \pmod{n_k}$.

31.6 PUISSANCES D'UN ÉLÉMENT

De même qu'il est naturel de considérer les multiples d'un élément a modulo n , de même il est naturel de considérer la séquence des puissances de a modulo n , où $a \in \mathbf{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \dots, \tag{31.29}$$

modulo n . En indexant à partir de 0, la 0^{ième} valeur de cette séquence est $a^0 \pmod{n}$ et la i ^{ème} est $a^i \pmod{n}$. Par exemple, les puissances de 3 modulo 7 sont

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$3^i \pmod{7}$	1	3	2	6	4	5	1	3	2	6	4	5	...

tandis que les puissances de 2 modulo 7 sont

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$2^i \pmod{7}$	1	2	4	1	2	4	1	2	4	1	2	4	...

Dans cette section, appelons $\langle a \rangle$ le sous-groupe de \mathbf{Z}_n^* généré par a et $\text{ord}_n(a)$ (l'« ordre de a modulo n ») l'ordre de a dans \mathbf{Z}_n^* . Par exemple, $\langle 2 \rangle = \{1, 2, 4\}$ dans \mathbf{Z}_7^* et $\text{ord}_7(2) = 3$. En utilisant la fonction d'Euler $\phi(n)$ pour la taille de \mathbf{Z}_n^* (voir section 31.3), on va reformuler le corollaire 31.19 avec la notation \mathbf{Z}_n^* pour obtenir le théorème d'Euler, que l'on va ensuite appliquer à \mathbf{Z}_p^* , avec p premier, pour obtenir le théorème de Fermat.

Théorème 31.30 (Théorème d'Euler) *Pour un entier $n > 1$ quelconque,*

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ pour tout } a \in \mathbf{Z}_n^* .$$

Théorème 31.31 (Théorème de Fermat) *Si p est premier,*

$$a^{p-1} \equiv 1 \pmod{p} \text{ pour tout } a \in \mathbf{Z}_p^*.$$

Démonstration : D'après l'équation (31.20), $\phi(p) = p - 1$ si p est premier. \square

Ce corollaire s'applique à tout élément de \mathbf{Z}_p sauf 0, puisque $0 \notin \mathbf{Z}_p^*$. Toutefois, pour tout $a \in \mathbf{Z}_p$, on a $a^p \equiv a \pmod{p}$ si p est premier.

Si $\text{ord}_n(g) = |\mathbf{Z}_n^*|$, alors tout élément de \mathbf{Z}_n^* est une puissance de g modulo n , et on dit que g est une **racine primitive** ou encore un **générateur** de \mathbf{Z}_n^* . Par exemple, 3 est une racine primitive modulo 7 mais 2 n'est pas une racine primitive modulo 7. Si \mathbf{Z}_n^* possède une racine primitive, on dit que le groupe \mathbf{Z}_n^* est **cyclique**. On trouvera la démonstration du théorème suivant, dû à Niven et Zuckerman, dans [231].

Théorème 31.32 *Les valeurs de $n > 1$ pour lesquelles \mathbf{Z}_n^* est cyclique sont 2, 4, p^e et $2p^e$, pour tout nombre premier p supérieur à 2 et pour tout entier strictement positif e .*

Si g est une racine primitive de \mathbf{Z}_n^* et si a est un élément quelconque de \mathbf{Z}_n^* , alors il existe un z tel que $g^z \equiv a \pmod{n}$. Ce z est appelé **logarithme discret** ou **indice** de a modulo n , dans la base g ; on représente cette valeur par $\text{ind}_{n,g}(a)$.

Théorème 31.33 (Théorème du logarithme discret) *Si g est une racine primitive de \mathbf{Z}_n^* , alors l'équation $g^x \equiv g^y \pmod{n}$ est satisfaite si et seulement si l'équation $x \equiv y \pmod{\phi(n)}$ l'est aussi.*

Démonstration : On commence par supposer que $x \equiv y \pmod{\phi(n)}$. Dans ce cas, $x = y + k\phi(n)$ pour un certain entier k . Donc,

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} \pmod{n} \\ &\equiv g^y \cdot (g^{\phi(n)})^k \pmod{n} \\ &\equiv g^y \cdot 1^k \pmod{n} \quad (\text{d'après le théorème d'Euler}) \\ &\equiv g^y \pmod{n}. \end{aligned}$$

Réciproquement, on suppose que $g^x \equiv g^y \pmod{n}$. Comme la séquence des puissances de g génère tous les éléments de $\langle g \rangle$ et comme $|\langle g \rangle| = \phi(n)$, le corollaire 31.18 implique que la séquence des puissances de g est périodique de période $\phi(n)$. Donc, si $g^x \equiv g^y \pmod{n}$, on doit avoir $x \equiv y \pmod{\phi(n)}$. \square

Le passage aux logarithmes discrets peut parfois simplifier le raisonnement dans une équation modulo, comme le montre la démonstration du théorème suivant.

Théorème 31.34 *Si p est un nombre premier impair et si $e \geq 1$, alors l'équation*

$$x^2 \equiv 1 \pmod{p^e} \tag{31.30}$$

n'a que deux solutions, à savoir $x = 1$ et $x = -1$.

Démonstration : Soit $n = p^e$. Le théorème 31.32 implique que \mathbf{Z}_n^* possède une racine primitive g . L'équation (31.30) peut alors s'écrire

$$(g^{\text{ind}_{n,g}(x)})^2 \equiv g^{\text{ind}_{n,g}(1)} \pmod{n}. \quad (31.31)$$

Après avoir remarqué que $\text{ind}_{n,g}(1) = 0$, on observe que le théorème 31.33 implique que l'équation (31.31) est équivalente à

$$2 \cdot \text{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)}. \quad (31.32)$$

Pour résoudre cette équation pour l'inconnue $\text{ind}_{n,g}(x)$, on applique les méthodes de la section 31.4. D'après l'équation (31.19), on a $\phi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$. En posant $d = \text{pgcd}(2, \phi(n)) = \text{pgcd}(2, (p - 1)p^{e-1}) = 2$ et en remarquant que $d \mid 0$, on trouve d'après le théorème 31.24 que l'équation (31.32) admet exactement $d = 2$ solutions. Donc, l'équation (31.30) admet exactement 2 solutions, qui sont $x = 1$ et $x = -1$ par inspection. \square

Un nombre x est un **racine carrée non triviale de 1, modulo n** , s'il vérifie l'équation $x^2 \equiv 1 \pmod{n}$ et si x n'est équivalent à aucune des deux racines carrées « triviales » 1 et -1 modulo n . Par exemple, 6 est une racine carrée de 1 non triviale, modulo 35. Le corollaire suivant du théorème 31.34 servira à démontrer la validité de la procédure Miller-Rabin de test de primalité, à la section 31.8.

Corollaire 31.35 *S'il existe une racine carrée non triviale de 1 modulo n , alors n est composé.*

Démonstration : D'après la contraposée du théorème 31.34, s'il existe une racine carrée non triviale de 1 modulo n , alors n ne peut être ni premier impair ni puissance d'un nombre premier impair. Si $x^2 \equiv 1 \pmod{2}$, alors $x \equiv 1 \pmod{2}$, de sorte que toutes les racines carrées de 1 modulo 2 sont triviales. Donc, n ne peut pas être premier. Enfin, on doit avoir $n > 1$ pour qu'il puisse exister une racine carrée non triviale de 1. Par conséquent, n est forcément composé. \square

a) Exponentiation par élévations répétées au carré

Une opération fréquente en théorie des nombres consiste à élever un nombre à une puissance modulo un autre nombre. On appelle cette opération **exponentiation modulaire**. Plus précisément, on souhaiterait disposer d'un moyen efficace de calculer $a^b \pmod{n}$, où a et b sont deux entiers positifs ou nuls et n est un entier strictement positif. L'exponentiation modulaire est une opération fondamentale pour de nombreuses routines de test de primalité, ainsi que pour le cryptosystème à clé publique RSA. La méthode des **élévations répétées au carré** résout ce problème efficacement en utilisant la représentation binaire de b .

Soit $\langle b_k, b_{k-1}, \dots, b_1, b_0 \rangle$ la représentation binaire de b . (Autrement dit, la représentation binaire a une longueur $k + 1$, b_k est le bit le plus significatif et b_0 est le bit le moins significatif.) La procédure suivante calcule $a^c \pmod{n}$ pour c croissant par doublements et incréments de 0 à b .

```

EXPONENTIATION-MODULAIRE( $a, b, n$ )
1   $c \leftarrow 0$ 
2   $d \leftarrow 1$ 
3  soit  $\langle b_k, b_{k-1}, \dots, b_0 \rangle$  la représentation binaire de  $b$ 
4  pour  $i \leftarrow k$  decr jusqu'à 0
5      faire  $c \leftarrow 2c$ 
6           $d \leftarrow (d \cdot d) \bmod n$ 
7          si  $b_i = 1$ 
8              alors  $c \leftarrow c + 1$ 
9               $d \leftarrow (d \cdot a) \bmod n$ 
10 retourner  $d$ 

```

L'utilisation fondamentale de l'élévation au carré en ligne 6 de chaque itération explique le nom « élévation répétée au carré ». À titre d'exemple, pour $a = 7$, $b = 560$ et $n = 561$, l'algorithme calcule la séquence de valeurs modulo 561 montrée à la figure 31.4 ; la suite des exposants employés est donnée sur la ligne étiquetée c de la table.

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Figure 31.4 Résultats de EXPONENTIATION-MODULAIRE pour le calcul de $a^b \pmod n$, avec $a = 7$, $b = 560 = \langle 1000110000 \rangle$ et $n = 561$. Les valeurs sont montrées après chaque exécution de la boucle **pour**. Le résultat final est 1.

La variable c n'est pas vraiment indispensable pour l'algorithme, mais elle permet de clarifier la procédure ; l'algorithme conserve l'invariant de boucle bipartite que voici : Juste avant chaque itération de la boucle **pour** des lignes 4-9,

- 1) La valeur de c est la même que le préfixe $\langle b_k, b_{k-1}, \dots, b_{i+1} \rangle$ de la représentation binaire de b , et
- 2) $d = a^c \bmod n$.

Nous utiliserons cet invariant de la façon suivante :

Initialisation : Initialement, $i = k$, de sorte que le préfixe $\langle b_k, b_{k-1}, \dots, b_{i+1} \rangle$ est vide, ce qui correspond à $c = 0$. De plus, $d = 1 = a^0 \bmod n$.

Conservation : Soient c' et d' les valeurs de c et d à la fin d'une itération de la boucle **pour**, et donc les valeurs avant l'itération suivante. Chaque itération fait $c' \leftarrow 2c$ (si $b_i = 0$) ou $c' \leftarrow 2c + 1$ (si $b_i = 1$), de sorte que c est correct avant la prochaine itération. Si $b_i = 0$, alors $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n$.

Si $b_i = 1$, alors $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. Dans l'un ou l'autre cas, $d = a^c \bmod n$ avant la prochaine itération.

Terminaison : À la fin de la boucle, $i = -1$. Donc, $c = b$ car c a la valeur du préfixe $\langle b_k, b_{k-1}, \dots, b_0 \rangle$ de la représentation binaire de b . Donc, $d = a^c \bmod n = a^b \bmod n$.

Si les entrées a , b et n sont des nombres de β bits, alors le nombre total d'opérations arithmétiques requises est $O(\beta)$ et le nombre total d'opérations binaires requises est $O(\beta^3)$.

Exercices

31.6.1 Dessiner une table montrant l'ordre de tous les éléments appartenant à \mathbf{Z}_{11}^* . Prendre la plus petite racine primitive g et calculer une table qui donne $\text{ind}_{11,g}(x)$ pour tout $x \in \mathbf{Z}_{11}^*$.

31.6.2 Donner un algorithme d'exponentiation modulaire qui examine les bits de b de la droite vers la gauche et non de la gauche vers la droite.

31.6.3 En supposant connu $\phi(n)$, montrer comment calculer $a^{-1} \bmod n$ pour tout $a \in \mathbf{Z}_n^*$ à l'aide de la procédure EXPONENTIATION-MODULAIRE.

31.7 LE CRYPTOSYSTÈME À CLÉS PUBLIQUES RSA

Un cryptosystème à clés publiques peut servir à chiffrer des messages échangés entre deux parties communicantes, pour qu'il soit impossible à une éventuelle oreille indiscreète de les décoder. Un cryptosystème à clés publiques permet aussi à l'une des parties d'ajouter une « signature numérique », impossible à contrefaire, à la fin d'un message électronique. Cette signature est la version électronique d'une signature manuscrite sur un document papier. Elle peut être facilement vérifiée par n'importe qui, personne ne peut la contrefaire, et pourtant elle perd sa validité dès qu'un bit du message est modifié. Elle permet ainsi une authentification à la fois de l'identité de l'expéditeur et du contenu du message. C'est l'outil parfait pour les contrats d'affaires signés électroniquement, les chèques électroniques, les ordres d'achat électroniques, et autres communications électroniques devant être authentifiées.

Le cryptosystème à clés publiques RSA est basé sur la différence énorme entre la facilité qu'il y a à trouver de grands nombres premiers et la difficulté qu'il y a à factoriser le produit de deux grands nombres premiers. La section 31.8 décrit une procédure efficace permettant de trouver des grands nombres premiers, et la section 31.9 étudie le problème de la factorisation de grands entiers.

a) Cryptosystèmes à clés publiques

Dans un cryptosystème à clés publiques, chaque participant possède à la fois une **clé publique** et une **clé secrète**. Chaque clé est une partie de l'information. Par exemple, dans le cryptosystème RSA, chaque clé est composée d'une paire d'entiers. Les participants « Alice » et « Bob » sont utilisés traditionnellement dans les exemples de cryptographie : leurs clés publique et secrète seront notées P_A , S_A pour Alice et P_B , S_B pour Bob.

Chaque participant crée ses propres clés publique et secrète. Chacun garde secrète sa clé secrète, mais il peut révéler sa clé publique à n'importe qui, et même la publier. En fait, il est souvent commode de supposer que la clé publique de tous est disponible dans un répertoire public, de façon que tout participant puisse facilement obtenir la clé publique de n'importe quel autre participant.

Les clés publique et secrète spécifient des fonctions qui peuvent être appliquées à n'importe quel message. Soit \mathcal{D} l'ensemble des messages possibles. Par exemple, \mathcal{D} pourrait être l'ensemble de toutes les séquences de bits de longueur finie. Dans la formulation originelle (et la plus simple) de la cryptographie à clé publique, il faut que les clés publique et secrète définissent des bijections de \mathcal{D} vers lui-même. La fonction correspondant à la clé publique d'Alice P_A est notée $P_A()$, et celle correspondant à sa clé secrète S_A est notée $S_A()$. Les fonctions $P_A()$ et $S_A()$ sont donc des permutations de \mathcal{D} . On suppose que les fonctions $P_A()$ et $S_A()$ sont calculables efficacement, une fois connue la clé P_A ou S_A correspondante.

Les clés publique et secrète de chaque participant forment un « couple », au sens où elles spécifient des fonctions qui sont inverses l'une de l'autre. Autrement dit,

$$M = S_A(P_A(M)) , \quad (31.33)$$

$$M = P_A(S_A(M)) \quad (31.34)$$

pour tout message $M \in \mathcal{D}$. La transformation de M à l'aide des deux clés P_A et S_A successivement, dans n'importe quel ordre, redonne le message M .

Dans un cryptosystème à clés publiques, il est impératif que personne d'autre qu'Alice ne puisse calculer la fonction $S_A()$ dans un temps raisonnable. Le caractère privé du courrier chiffré et envoyé à Alice, ainsi que l'authenticité des signatures numériques d'Alice, reposent sur l'hypothèse que seule Alice est capable de calculer $S_A()$. C'est pour cela qu'Alice tient secrète sa clé S_A ; dans le cas contraire, elle perd son unicité, et le cryptosystème ne peut plus lui garantir des fonctionnalités qui lui sont propres. L'hypothèse selon laquelle Alice est la seule à pouvoir calculer $S_A()$ doit être satisfaite même si tout le monde connaît P_A et est capable de calculer $P_A()$, fonction inverse de $S_A()$, efficacement. La grande difficulté, lors de l'élaboration d'un cryptosystème à clés publiques opérationnel, est de concevoir un système permettant de révéler une transformation $P_A()$ sans pour autant révéler la manière dont la transformation inverse $S_A()$ se calcule.

Dans un cryptosystème à clés publiques, le chiffrement fonctionne de la manière suivante. Supposons que Bob souhaite envoyer à Alice un message M chiffré, de sorte qu'il ait l'apparence d'un charabia incompréhensible pour une oreille indiscreète. Le scénario pour l'envoi du message est le suivant.

- Bob se procure la clé publique d'Alice P_A (à partir d'un répertoire public, ou par Alice directement).
- Bob calcule le **texte chiffré** $C = P_A(M)$ correspondant au message M et envoie C à Alice.
- Quand Alice reçoit le texte chiffré C , elle lui applique sa clé secrète S_A pour lire le message original : $M = S_A(C)$.

Comme $S_A()$ et $P_A()$ sont des fonctions inverses, Alice peut calculer M à partir de C . Comme seule Alice est capable de calculer $S_A()$, elle seule peut calculer M à partir de C . Le chiffrement de M à l'aide de $P_A()$ a permis à M de n'être déchiffré par personne excepté Alice.

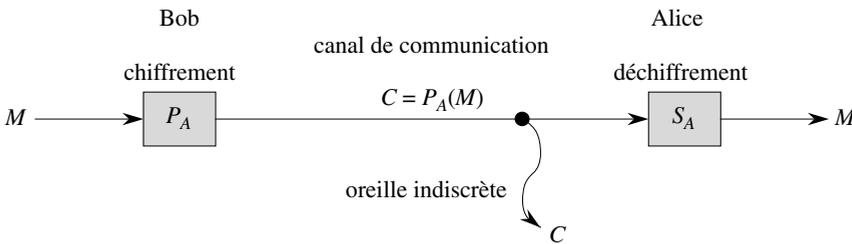


Figure 31.5 Chiffrement dans un système à clés publiques. Bob chiffre le message M à l'aide de la clé publique d'Alice P_A et transmet le message chiffré résultant $C = P_A(M)$ à Alice. Une oreille indiscreète qui détournerait le message chiffré transmis ne peut obtenir aucune information sur M . Alice reçoit C et le déchiffre à l'aide de sa clé secrète, pour obtenir le message original $M = S_A(C)$.

Il est tout aussi simple d'implémenter des signatures numériques dans un tel cryptosystème à clés publiques. (Il existe d'autres méthodes pour créer des signatures numériques, mais ceci est une autre histoire.) Supposons à présent qu'Alice souhaite envoyer à Bob une réponse M' signée. Le processus de signature numérique fonctionne de la manière suivante, illustrée à la figure 31.6.

- Alice calcule sa **signature numérique** σ pour le message M' à l'aide de sa clé secrète S_A et de l'équation $\sigma = S_A(M')$.
- Alice envoie la paire message-signature (M', σ) à Bob.
- Lorsque Bob reçoit (M', σ) , il peut vérifier qu'il vient bien d'Alice en utilisant la clé publique d'Alice pour vérifier l'équation $M' = P_A(\sigma)$. (On peut supposer que M' contient le nom d'Alice, pour que Bob sache quelle clé publique utiliser.) Si l'équation est vérifiée, alors Bob en déduit que le message M' a effectivement été

signé par Alice. Dans le cas contraire, Bob en déduit soit que le message M' ou la signature numérique σ a été corrompu par des erreurs de transmission, soit que la paire (M', σ) est une tentative de faux.

Puisqu'une signature numérique authentifie et l'identité de l'expéditeur et le contenu du message signé, elle équivaut à une signature manuscrite au bas d'un document écrit.

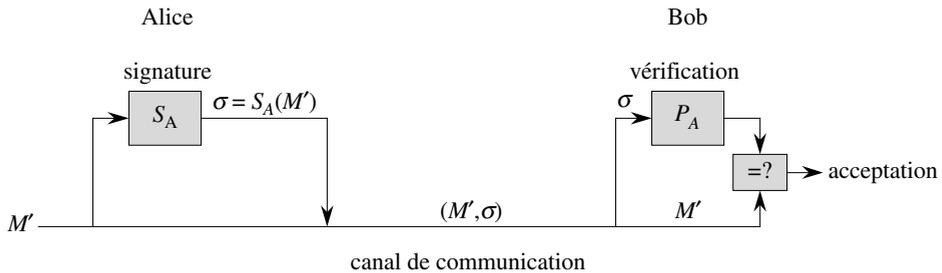


Figure 31.6 Signatures numériques dans un système à clés publiques. Alice signe le message M' en y ajoutant sa signature numérique $\sigma = S_A(M')$. Elle transmet la paire message-signature (M', σ) à Bob, qui peut la tester en vérifiant l'équation $M' = P_A(\sigma)$. Si l'équation est satisfaite, il accepte (M', σ) comme message signé par Alice.

Une propriété importante de la signature numérique est qu'elle est vérifiable par n'importe qui ayant accès à la clé publique du signataire. Un message signé peut être vérifié par une partie, et ensuite passé à d'autres parties, qui peuvent à leur tour vérifier la signature. Par exemple, le message pourra être un chèque électronique d'Alice à Bob. Après vérification par Bob de la signature d'Alice, il peut donner le chèque à sa banque, qui pourra également vérifier la signature et effectuer le virement approprié.

Notons qu'un message signé n'est pas chiffré ; le message est envoyé « en clair » et n'est pas protégé. En combinant les deux protocoles précédents qui concernent respectivement le chiffrement et la signature, on peut créer des messages à la fois signés et chiffrés. L'expéditeur commence par joindre sa signature numérique au message, puis chiffre la paire message-signature résultante avec la clé publique du destinataire. Celui-ci déchiffre le message avec sa clé secrète pour obtenir à la fois le message original et sa signature numérique. Il peut alors vérifier la signature à l'aide de la clé publique du signataire. Quand on utilise les protocoles basés sur l'échange de documents papier, le processus correspondant consiste à signer le document, puis à glisser le document dans une enveloppe scellée qui ne sera ouverte que par le destinataire.

b) Le cryptosystème RSA

Dans le **cryptosystème à clés publiques RSA**, un participant crée ses clés publique et secrète en suivant la procédure suivante.

- 1) Choisir aléatoirement deux grands nombres premiers p et q différents. Les nombres p et q feront, par exemple, 512 bits chacun.
- 2) Calculer n à partir de l'équation $n = pq$.
- 3) Choisir un petit entier impair e premier avec $\phi(n)$ qui, d'après l'équation (31.19), est égal à $(p - 1)(q - 1)$.
- 4) Calculer d , inverse de e modulo $\phi(n)$. (Le corollaire 31.26 assure que d existe et est unique. On peut employer la technique de la section 31.4 pour calculer d , à partir de e et $\phi(n)$.)
- 5) Publier la paire $P = (e, n)$ qui sera sa **clé publique RSA**.
- 6) Garder secrète la paire $S = (d, n)$ qui sera sa **clé secrète RSA**.

Pour ce modèle, le domaine \mathcal{D} est l'ensemble \mathbf{Z}_n . La transformation d'un message M associé à la clé publique $P = (e, n)$ est

$$P(M) = M^e \pmod{n} . \quad (31.35)$$

La transformation d'un texte chiffré C associé à une clé secrète $S = (d, n)$ est

$$S(C) = C^d \pmod{n} . \quad (31.36)$$

Ces équations s'appliquent à la fois au chiffrement et aux signatures. Pour créer une signature, l'expéditeur applique sa clé secrète au message à signer, et non à un texte chiffré. Pour vérifier une signature, on applique la clé publique du signataire à la signature et non à un message à chiffrer.

Les opérations de clé publique et de clé secrète peuvent être implémentées à l'aide de la procédure EXPONENTIATION-MODULAIRE décrite à la section 31.6. Pour analyser le temps d'exécution de ces opérations, on suppose que la clé publique (e, n) et la clé secrète (d, n) vérifient les égalités $\lg e = O(1)$, $\lg d \leq \beta$ et $\lg n \leq \beta$. Alors, l'application d'une clé publique nécessite $O(1)$ multiplications modulaires et emploie $O(\beta^2)$ opérations de bits. L'application d'une clé secrète nécessite $O(\beta)$ multiplications modulaires, utilisant $O(\beta^3)$ opérations de bits.

Théorème 31.36 (Validité de RSA) *Les équations RSA (31.35) et (31.36) définissent des transformations inverses de \mathbf{Z}_n qui vérifient les équations (31.33) et (31.34).*

Démonstration : En partant des équations (31.35) et (31.36), on a, pour tout $M \in \mathbf{Z}_n$,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n} .$$

Comme e et d sont inverses modulo $\phi(n) = (p - 1)(q - 1)$,

$$ed = 1 + k(p - 1)(q - 1)$$

pour un certain entier k . Mais dans ce cas, si $M \not\equiv 0 \pmod{p}$, on a

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} \pmod{p} \\ &\equiv M(1)^{k(q-1)} \pmod{p} \quad (\text{d'après le théorème 31.31}) \\ &\equiv M \pmod{p} . \end{aligned}$$

D'autre part, $M^{ed} \equiv M \pmod{p}$ si $M \equiv 0 \pmod{p}$. Donc,

$$M^{ed} \equiv M \pmod{p}$$

pour tout M . De même,

$$M^{ed} \equiv M \pmod{q}$$

pour tout M . Donc, d'après le corollaire 31.29 du théorème du reste Chinois,

$$M^{ed} \equiv M \pmod{n}$$

pour tout M . □

La sécurité du cryptosystème RSA repose en grande partie sur la difficulté de factoriser les grands entiers. Si une personne mal intentionnée parvient à factoriser le module n d'une clé publique, elle peut en déduire la clé secrète, en utilisant les facteurs p et q de la même manière que le créateur de la clé publique. Donc, si la factorisation des grands entiers est facile, il est facile de casser le cryptosystème RSA. La proposition inverse « La complexité de la factorisation des grands entiers implique-t-elle la complexité du cassage de RSA ? » n'a pas été prouvée. Toutefois, après deux décennies de recherches, personne n'a trouvé de moyen plus simple pour casser le système RSA que de factoriser le module n . Et comme nous le verrons à la section 31.9, la factorisation des grands entiers est étonnamment difficile. En choisissant aléatoirement puis en multipliant deux nombres premiers de 512 bits, on peut créer une clé publique impossible à « casser » en un temps raisonnable avec la technologie actuelle. À moins d'une percée fondamentale dans la conception d'algorithmes de théorie des nombres, le cryptosystème RSA fournit un haut degré de sécurité aux applications, pour peu qu'on l'implémente avec soin en suivant des normes recommandées.

Pour que la sécurité soit vraiment bonne, il convient toutefois de travailler avec des entiers faisant plusieurs centaines de bits, afin de faire face à des avancées possibles en matière de factorisation. À l'heure où nous écrivons ces lignes (2001), les modules n de RSA font généralement de 768 à 2048 bits. Créer des modules d'une telle taille exige que l'on soit capable de trouver efficacement de grands nombres premiers. Ce problème est étudié à la section 31.8.

Pour des questions de rapidité, on utilise souvent RSA en mode « hybride » ou « gestion de clés » avec des cryptosystèmes rapides à clés non publiques. Avec un tel système, les clés de chiffrement et de déchiffrement sont identiques. Si Alice souhaite envoyer un long message M à Bob de façon confidentielle, elle choisit une clé aléatoire K pour le cryptosystème rapide à clés non publiques et chiffre M avec K pour obtenir le texte chiffré C . Ici, C est aussi long que M , mais K est très courte. Ensuite, elle chiffre K avec la clé publique RSA de Bob. Comme K est courte, le calcul de $P_B(K)$ est rapide (beaucoup plus rapide que celui de $P_B(M)$). Elle transmet alors $(C, P_B(K))$ à Bob, qui déchiffre $P_B(K)$ pour obtenir K puis utilise K pour déchiffrer C , obtenant ainsi M .

Une approche hybride similaire est souvent utilisée pour créer rapidement des signatures numériques. Dans cette approche, on combine RSA avec une *fonction de*

hachage anti-collision h , c'est-à-dire une fonction facile à calculer mais pour laquelle il est calculatoirement impossible de trouver deux messages M et M' tels que $h(M) = h(M')$. La valeur $h(M)$ est une petite (disons, 160 bits) « empreinte digitale » du message M . Si Alice veut signer un message M , elle commence par appliquer h à M pour obtenir l'empreinte $h(M)$, qu'elle chiffre ensuite avec sa clé secrète. Elle envoie $(M, S_A(h(M)))$ à Bob, comme étant sa version signée de M . Bob peut vérifier la signature en calculant $h(M)$ et en vérifiant que P_A appliqué à $S_A(h(M))$ (tel que reçu) donne $h(M)$. Comme personne ne peut créer deux messages ayant la même empreinte digitale, il est calculatoirement infaisable d'altérer un message signé tout en préservant la validité de la signature.

Enfin, notons que l'utilisation de **certificats** rend la distribution des clés publiques beaucoup plus simple. Par exemple, supposons qu'il existe une « autorité de confiance » T dont la clé publique est connue de tous. Alice peut demander à T un message signé (son certificat) établissant que « la clé publique d'Alice est P_A ». Ce certificat est « auto-authentifiable » puisque tout le monde connaît P_T . Alice peut inclure son certificat avec ses messages signés, de sorte que le destinataire peut connaître immédiatement la clé publique d'Alice pour vérifier sa signature. Comme sa clé a été signée par T , le destinataire sait que la clé d'Alice est bien celle d'Alice.

Exercices

31.7.1 Considérons un ensemble de clés RSA avec $p = 11$, $q = 29$, $n = 319$ et $e = 3$. Quel valeur de d devrait-on utiliser dans la clé secrète ? Quel est le résultat chiffré du message $M = 100$?

31.7.2 Démontrer que si l'exposant publique d'Alice e vaut 3 et qu'un espion obtient l'exposant secret d d'Alice, il peut factoriser le modulo n d'Alice en un temps qui est polynomial par rapport au nombre de bits de n . (Bien que la démonstration n'en soit pas demandée, il est intéressant de savoir que ce résultat reste vrai même si la condition $e = 3$ est supprimée. Voir Miller [221].)

31.7.3 ★ Démontrer que RSA est multiplicatif au sens où

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}.$$

Utiliser ce fait pour montrer que, si un adversaire dispose d'une procédure capable de déchiffrer efficacement 1 pour cent des messages de \mathbf{Z}_n chiffrés via P_A , alors il pourrait employer un algorithme probabiliste pour déchiffrer avec une grande probabilité tous les messages chiffrés via P_A .

31.8 ★ TEST DE PRIMARITÉ

Dans cette section, on considère le problème consistant à trouver de grands nombres premiers. On commence avec une étude de la densité des nombres premiers, puis on

examine une approche plausible (mais incomplète) du test de primalité, et enfin on présente un test de primalité randomisé efficace dû à Miller et Rabin.

a) Densité des nombres premiers

Pour de nombreuses applications (comme la cryptographie), il est utile de trouver de grands nombres premiers « aléatoires ». Heureusement, les grands nombres premiers ne sont pas trop rares, et il n'est pas trop coûteux de tester des entiers aléatoires de la taille désirée jusqu'à trouver un nombre premier. La **fonction de distribution des nombres premiers** $\pi(n)$ spécifie la quantité de nombres premiers inférieurs ou égaux à n . Par exemple, $\pi(10) = 4$, car il existe 4 nombres premiers inférieurs ou égaux à 10, à savoir 2, 3, 5 et 7. Le théorème des nombres premiers donne une approximation utile de $\pi(n)$.

Théorème 31.37 (Théorème des nombres premiers)

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \ln n} = 1 .$$

L'approximation $n / \ln n$ donne une estimation raisonnable de $\pi(n)$, même pour des n petits. Par exemple, elle fait une erreur de moins de 6% pour $n = 10^9$, où $\pi(n) = 50\,847\,534$ et $n / \ln n = 48\,254\,942$. (Pour un théoricien des nombres, 10^9 est un petit nombre.)

On peut utiliser le théorème des nombres premiers pour estimer à $1 / \ln n$ la probabilité qu'un entier n choisi aléatoirement soit premier. Donc, il faudra examiner environ $\ln n$ entiers choisis aléatoirement autour de n pour trouver un nombre premier de la même longueur que n . Par exemple, trouver un nombre premier de 512 bits demande de tester environ $\ln 2^{512} \approx 355$ nombres de 512 bits choisis aléatoirement. (On peut réduire de moitié cette quantité en ne prenant que des entiers impairs.)

Dans le reste de cette section, on considère le problème consistant à déterminer si un grand entier impair n est premier. Par commodité notationnelle, on suppose que n a une décomposition en facteurs premiers de la forme

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \quad (31.37)$$

où $r \geq 1$ et p_1, p_2, \dots, p_r sont les facteurs premiers de n . Bien sûr, n est premier si et seulement si $r = 1$ et $e_1 = 1$.

Une première approche du test de primalité consiste à faire des **essais de division**. On tente de diviser n par chaque entier compris entre 2 et $\lfloor \sqrt{n} \rfloor$. (Ici encore, on pourra sauter les entiers pairs supérieurs à 2.) On voit aisément que n n'est premier que si aucun des diviseurs tentés ne divise n . En supposant que chaque essai de division nécessite un temps constant, le temps d'exécution dans le pire des cas est $\Theta(\sqrt{n})$, qui est exponentiel par rapport à la longueur de n . (Rappelez-vous que, si n est représenté en binaire par β bits, alors $\beta = \lceil \lg(n+1) \rceil$, et donc $\sqrt{n} = \Theta(2^{\beta/2})$.) Donc, ces divisions répétées ne sont efficaces que pour des n très petits ou qui possèdent un

petit facteur premier. Quand elle fonctionne, la méthode des essais de division a pour avantage non seulement de déterminer si n est premier ou non, mais aussi de fournir l'un de ses facteurs premiers au cas où n est composé.

Dans cette section, nous cherchons uniquement à savoir si un nombre n donné est premier ; si n est composé, nous n'avons aucun besoin de trouver sa décomposition en facteurs premiers. Comme nous le verrons à la section 31.9, le calcul de la décomposition en nombres premiers est très coûteux calculatoirement. On sera peut-être surpris d'apprendre qu'il est beaucoup plus facile de dire si un nombre donné est premier ou non, que de déterminer sa décomposition en facteurs premiers si ce nombre est composé.

b) Test de pseudo primarité

On va maintenant étudier un test de primarité qui « fonctionne presque » et qui est, en fait, assez bon pour de nombreuses applications pratiques. Un affinement de cette méthode, qui supprime la petite imperfection, sera présenté plus tard. Soit \mathbf{Z}_n^+ l'ensemble des éléments non nuls de \mathbf{Z}_n :

$$\mathbf{Z}_n^+ = \{1, 2, \dots, n-1\} .$$

Si n est premier, alors $\mathbf{Z}_n^+ = \mathbf{Z}_n^*$.

On dit que n est un nombre *pseudo premier de base a* si n est un nombre composé et

$$a^{n-1} \equiv 1 \pmod{n} . \quad (31.38)$$

Le théorème de Fermat implique que, si n est premier, il vérifie l'équation (31.38) pour tout a de \mathbf{Z}_n^+ . Donc, si l'on peut trouver un $a \in \mathbf{Z}_n^+$ tel que n ne vérifie pas l'équation (31.38), alors n est forcément composé. Étonnamment, la réciproque est *presque toujours* vraie, de sorte que ce critère forme un test de primarité quasi parfait. On teste si n vérifie l'équation (31.38) pour $a = 2$. Si tel n'est pas le cas, on déclare que n est composé. Autrement, on fait l'hypothèse que n est premier (alors qu'en fait on sait seulement que n est soit premier, soit pseudo premier de base 2).

La procédure suivante fait semblant, de cette manière, de tester la primarité de n . Elle fait appel à la procédure EXPONENTIATION-MODULAIRE de la section 31.6. On suppose que l'entrée n est entière, impaire et supérieure à 2.

PSEUDO-PREMIER(n)

- 1 **si** EXPONENTIATION-MODULAIRE($2, n-1, n$) $\not\equiv 1 \pmod{n}$
- 2 **alors retourner** COMPOSÉ ▷ c'est sûr !
- 3 **sinon retourner** PREMIER ▷ Avec un peu de chance !

Cette procédure peut faire des erreurs, mais d'un seul type. Si elle dit que n est composé, c'est vrai. En revanche, si elle affirme que n est premier, elle fait une erreur dans le cas où n est pseudo premier de base 2.

Combien de chances cette procédure a-t-elle de se tromper ? Bizarrement, très peu. Il n'existe que 22 valeurs de n inférieures à 10 000 pour lesquelles elle se trompe ; les quatre premières sont 341, 561, 645 et 1105. On peut montrer que la probabilité pour que ce programme fasse une erreur sur un entier de β bits choisi arbitrairement tend vers zéro quand $\beta \rightarrow \infty$. En utilisant une estimation plus précise due à Pomerance [244] du nombre de pseudo premiers de base-2 d'une taille donnée, on peut estimer qu'un nombre de 512 bits choisi aléatoirement et déclaré premier par la procédure précédente a moins d'une chance sur 10^{20} d'être un pseudo premier de base 2, tandis qu'un nombre de 1 024 bits choisi aléatoirement et déclaré premier a moins d'une chance sur 10^{41} d'être un pseudo premier de base 2. Si donc vous essayez tout simplement de trouver un grand nombre premier pour une certaine application, dans tous les cas pratiques vous ne risquez rien à choisir de grands nombres au hasard jusqu'à ce que l'un d'eux entraîne PSEUDO-PREMIER à afficher PREMIER. Mais quand les nombres testés ne sont pas choisis aléatoirement, il faut une meilleure approche pour tester la primalité. Nous verrons qu'un peu d'astuce, combiné à un peu de randomisation, permet d'obtenir une routine de test de primalité qui marche bien sur toutes les entrées. Malheureusement, on ne peut pas éliminer toutes les erreurs en se contentant de tester l'équation (31.38) pour une deuxième base, $a = 3$ par exemple, car il existe des entiers composés n qui vérifient l'équation (31.38) pour *tout* $a \in \mathbf{Z}_n^*$. Ces entiers s'appellent des **nombre de Carmichael**. Les trois premiers nombres de Carmichael sont 561, 1105 et 1729. Ces nombres sont extrêmement rares ; il n'en existe par exemple que 255 inférieurs à 100 000 000. L'exercice 31.8.2 aide à comprendre pourquoi ils sont si rares.

Nous allons maintenant montrer comment notre test de primalité peut être amélioré pour ne pas être induit en erreur par les nombres de Carmichael.

c) Test de primalité randomisé de Miller-Rabin

Le test de primalité de Miller-Rabin surmonte les problèmes inhérents au test PSEUDO-PREMIER, en lui apportant deux modifications :

- Il essaye plusieurs valeurs de base a choisies aléatoirement, et non plus une seule.
- Pendant qu'il calcule chaque exponentiation modulaire, il regarde si une racine carrée non triviale de 1 modulo n est découverte pendant l'ensemble final d'élévations au carré. Dans ce cas, il s'arrête et affiche COMPOSÉ. Le corollaire 31.35 justifie la détection des nombres composés de cette manière.

Le pseudo code du test de primalité de Miller-Rabin est donné ci-après. L'entrée $n > 2$ est le nombre impair à tester, et s est le nombre de valeurs de base choisies aléatoirement dans \mathbf{Z}_n^+ qu'il faudra essayer. Le code fait appel au générateur de nombres aléatoires RANDOM (voir page 90) : RANDOM(1, $n - 1$) retourne un entier a choisi arbitrairement et vérifiant $1 \leq a \leq n - 1$. Le code utilise une procédure auxiliaire TÉMOIN telle que TÉMOIN(a, n) vaut VRAI si et seulement si a est un « témoin » du caractère composé de n , autrement dit s'il est possible de prouver à l'aide

de a (d'une façon que nous allons voir) que n est composé. Le test TÉMOIN(a, n) est une extension (plus efficace) du test

$$a^{n-1} \not\equiv 1 \pmod{n}$$

qui formait la base (pour $a = 2$) de PSEUDO-PREMIER. On commencera par présenter et justifier la construction de TÉMOIN, puis on montrera comment elle est utilisée dans le test de primalité de Miller-Rabin. Soit $n - 1 = 2^t u$ où $t \geq 1$ et u impair ; c'est-à-dire que la représentation binaire de $n - 1$ est la représentation binaire de l'entier impair u , suivie d'exactly t zéros. Donc, $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$, de sorte que l'on peut calculer $a^{n-1} \pmod{n}$ en calculant d'abord $a^u \pmod{n}$ puis en élevant au carré le résultat t fois d'affilée.

TÉMOIN(a, n)

```

1  soit  $n - 1 = 2^t u$ , avec  $t \geq 1$  et  $u$  impair
2   $x_0 \leftarrow$  EXPONENTIATION-MODULAIRE( $a, u, n$ )
3  pour  $i \leftarrow 1$  à  $t$ 
4      faire  $x_i \leftarrow x_{i-1}^2 \pmod{n}$ 
5          si  $x_i = 1$  et  $x_{i-1} \neq 1$  et  $x_{i-1} \neq n - 1$ 
6              alors retourner VRAI
7  si  $x_t \neq 1$ 
8      alors retourner VRAI
9  retourner FAUX

```

Ce pseudo code de TÉMOIN calcule $a^{n-1} \pmod{n}$ en calculant d'abord la valeur $x_0 = a^u \pmod{n}$ en ligne 2, puis en élevant au carré le résultat t fois d'affilée dans la boucle **pour** des lignes 3–6. Par récurrence sur i , la séquence x_0, x_1, \dots, x_t des valeurs calculées vérifie l'équation $x_i \equiv a^{2^i u} \pmod{n}$ pour $i = 0, 1, \dots, t$; et donc, on a en particulier $x_t \equiv a^{n-1} \pmod{n}$. Chaque fois qu'une étape d'élévation au carré est faite en ligne 4, la boucle peut cependant se terminer prématurément si les lignes 5–6 détectent qu'une racine carrée non triviale de 1 vient d'être découverte. Si c'est le cas, l'algorithme s'arrête et retourne VRAI. Les lignes 7–8 retournent VRAI si la valeur calculée pour $x_t \equiv a^{n-1} \pmod{n}$ n'est pas égale à 1, tout comme la procédure PSEUDO-PREMIER retourne COMPOSÉ dans ce cas. La ligne 9 retourne FAUX si l'on n'a pas retourné VRAI en ligne 6 ou 8.

Montrons à présent que, si TÉMOIN(a, n) retourne VRAI, alors on peut construire à l'aide de a une preuve de la nature composée de n .

Si TÉMOIN retourne VRAI depuis la ligne 8, c'est qu'elle a découvert que $x_t = a^{n-1} \pmod{n} \neq 1$. Si n est premier, en revanche, on sait d'après le théorème de Fermat que $a^{n-1} \equiv 1 \pmod{n}$ pour tout $a \in \mathbf{Z}_n^+$. Donc, n ne peut pas être premier et l'équation $a^{n-1} \pmod{n} \neq 1$ en est une preuve.

Si TÉMOIN retourne VRAI depuis la ligne 6, c'est qu'elle a découvert que x_{i-1} est une racine carrée non triviale de $x_i = 1$ modulo n , car on a $x_{i-1} \not\equiv \pm 1 \pmod{n}$ mais $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$. Le corollaire 31.35 dit que ce n'est que si n est composé

qu'il peut y avoir une racine carrée non triviale de 1 modulo n ; de sorte que démontrer que x_{i-1} est une racine carrée non triviale de 1 modulo n revient à démontrer que n est composé.

Cela termine notre démonstration de la validité de TÉMOIN. Si l'on obtient la valeur VRAI après l'appel TÉMOIN(a, n), on peut être certain que n est composé, et on peut facilement trouver une preuve de la nature composée de n à partir de a et n . À ce stade, nous allons présenter brièvement une autre description du comportement de TÉMOIN comme fonction de la séquence $X = \langle x_0, x_1, \dots, x_t \rangle$, ce qui pourra vous aider plus tard, quand nous analyserons l'efficacité du test de Miller-Rabin. Notez que, si $x_i = 1$ pour un certain $0 \leq i < t$, alors TÉMOIN risque de ne pas calculer le reste de la séquence. Si cela devait se produire, toutefois, chaque valeur $x_{i+1}, x_{i+2}, \dots, x_t$ serait 1 et l'on traiterait ces emplacements dans la séquence X comme étant tous des 1. On a quatre cas :

- 1) $X = \langle \dots, d \rangle$, avec $d \neq 1$: la séquence X ne finit pas par 1. On retourne VRAI ; a est un témoin du caractère composé de n (en vertu de Fermat).
- 2) $X = \langle 1, 1, \dots, 1 \rangle$: la séquence X n'est faite que de 1. On retourne FAUX ; a n'est pas un témoin du caractère composé de n .
- 3) $X = \langle \dots, -1, 1, \dots, 1 \rangle$: la séquence X finit par 1, et la dernière valeur non-1 est -1 . On retourne FAUX ; a n'est pas un témoin du caractère composé de n .
- 4) $X = \langle \dots, d, 1, \dots, 1 \rangle$, avec $d \neq \pm 1$: la séquence X finit par 1, mais la dernière valeur non-1 n'est pas -1 . On retourne VRAI ; a est un témoin du caractère composé de n , car d est une racine carrée non triviale de 1.

Examinons maintenant le test de primalité de Miller-Rabin basé sur l'utilisation de TÉMOIN. Ici aussi, nous supposons que n est un entier impair supérieur à 2.

MILLER-RABIN(n, s)

- 1 **pour** $j \leftarrow 1$ à s
- 2 **faire** $a \leftarrow \text{RANDOM}(1, n - 1)$
- 3 **si** TÉMOIN(a, n)
- 4 **alors retourner** COMPOSÉ ▷ c'est sûr !
- 5 **retourner** PREMIER ▷ c'est à peu près sûr.

La procédure MILLER-RABIN est une recherche probabiliste de la preuve que n est composé. La boucle principale (qui commence à la ligne 1), prend s valeurs aléatoires de a dans \mathbf{Z}_n^+ (ligne 2). Si l'un des a est témoin du caractère composé de n , alors MILLER-RABIN affiche COMPOSÉ à la ligne 4. Ce résultat est toujours correct, puisque nous avons établi que TÉMOIN était correct. Si aucun témoin ne peut être trouvé en s essais, MILLER-RABIN suppose qu'on ne pourra pas en trouver, et donc que n est premier. Nous verrons que ce résultat a de fortes chances d'être correct si s est assez grand, mais qu'il reste encore une petite chance pour que la procédure soit malchanceuse dans ses choix pour a et qu'il existe des témoins, bien qu'aucun n'ait été trouvé.

À titre d'exemple de l'action de MILLER-RABIN, donnons à n la valeur du nombre de Carmichael 561, de sorte que $n - 1 = 560 = 2^4 \cdot 35$. En supposant que $a = 7$ soit choisi comme base, la figure 31.4 montre que TÉMOIN calcule $x_0 \equiv a^{35} \equiv 241 \pmod{561}$ et donc calcule la séquence $X = \langle 241, 298, 166, 67, 1 \rangle$. Donc, il y a découverte d'une racine carrée non triviale de 1 lors de la dernière étape d'élévation au carré, vu que $a^{280} \equiv 67 \pmod{n}$ et $a^{560} \equiv 1 \pmod{n}$. Donc, $a = 7$ est un témoin du caractère composé de n , TÉMOIN(7, n) retourne VRAI et MILLER-RABIN retourne COMPOSÉ.

Si n est un nombre à β bits, MILLER-RABIN nécessite $O(s\beta)$ opérations arithmétiques et $O(s\beta^3)$ opérations de bits, puisqu'elle n'est pas plus complexe asymptotiquement que s exponentiations modulaires.

d) Taux d'erreur du test de primarité de Miller-Rabin

Si MILLER-RABIN affiche PREMIER, il existe un petit risque pour qu'elle ait fait une erreur. Toutefois, et contrairement à PSEUDO-PREMIER, les risques d'erreur ne dépendent pas de n ; il n'existe pas de mauvaise entrée pour cette procédure. Le risque dépend plutôt de la taille de s et du caractère « chanceux » du tirage des valeurs de bases a . Par ailleurs, comme chaque test est plus contraignant qu'une simple vérification de l'équation (31.38), on peut tabler sur des principes généraux pour dire que le taux d'erreur sera faible pour des entiers n choisis aléatoirement. Le théorème suivant présente un argument plus précis.

Théorème 31.38 *Si n est un nombre composé impair, alors le nombre de témoins de sa nature composée est au moins égal à $(n - 1)/2$.*

Démonstration : La démonstration montrera que le nombre de non-témoins ne dépasse pas $(n - 1)/2$, ce qui impliquera le théorème.

On commence par affirmer qu'un non-témoin doit appartenir à \mathbf{Z}_n^* . Pourquoi ? Un non-témoin a doit vérifier $a^{n-1} \equiv 1 \pmod{n}$, ou, ce qui revient au même, $a \cdot a^{n-2} \equiv 1 \pmod{n}$. Donc, il y a une solution pour l'équation $ax \equiv 1 \pmod{n}$, à savoir a^{n-2} . D'après le corollaire 31.21, $\text{pgcd}(a, n) \mid 1$, ce qui entraîne en retour que $\text{pgcd}(a, n) = 1$. Par conséquent, a est un membre de \mathbf{Z}_n^* ; tous les non témoins appartiennent à \mathbf{Z}_n^* .

Pour compléter la démonstration, montrons que non seulement tous les non témoins sont contenus dans \mathbf{Z}_n^* , mais qu'ils sont tous contenus dans un sous-groupe propre B de \mathbf{Z}_n^* (B est un sous-groupe propre de \mathbf{Z}_n^* si B est un sous-groupe de \mathbf{Z}_n^* sans être égal à \mathbf{Z}_n^*). D'après le corollaire 31.16, on a alors $|B| \leq |\mathbf{Z}_n^*|/2$. Comme $|\mathbf{Z}_n^*| \leq n - 1$, on a $|B| \leq (n - 1)/2$. Donc, le nombre de non témoins est au plus égal à $(n - 1)/2$, de sorte que le nombre de témoins est au moins égal à $(n - 1)/2$.

Nous allons maintenant montrer comment trouver un sous-groupe propre B de \mathbf{Z}_n^* contenant tous les non témoins. La preuve est divisée en deux parties.

Cas 1 : Il existe un $x \in \mathbf{Z}_n^*$ tel que

$$x^{n-1} \not\equiv 1 \pmod{n}.$$

Autrement dit, n n'est pas un nombre de Carmichael. Comme les nombres de Carmichael sont très rares (comme précédemment mentionné), le cas 1 est le cas courant « en pratique » (c'est-à-dire, quand n a été choisi au hasard et qu'on teste sa primalité). Soit $B = \{b \in \mathbf{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. B est visiblement non vide, vu qu'il contient 1. Comme B est fermé pour la multiplication modulo n , on sait que B est un sous-groupe de \mathbf{Z}_n^* d'après le théorème 31.14. Notez que tous les non témoins appartiennent à B , puisqu'un non témoin a vérifie $a^{n-1} \equiv 1 \pmod{n}$. Comme $x \in \mathbf{Z}_n^* - B$, B est un sous-groupe propre de \mathbf{Z}_n^* .

Cas 2 : Pour tout $x \in \mathbf{Z}_n^*$,

$$x^{n-1} \equiv 1 \pmod{n}. \quad (31.39)$$

Autrement dit, n est un nombre de Carmichael. Ce cas est très rare en pratique. Toutefois, le test de Miller-Rabin (contrairement au test de pseudo primalité) peut déterminer efficacement le caractère composé des nombres de Carmichael, comme nous allons le montrer.

Ici, n ne peut pas être une puissance d'un nombre premier. Pour comprendre pourquoi, posons $n = p^e$ où p est premier et $e > 1$. Nous allons aboutir à une contradiction. En effet, comme n est supposé impair, p est forcément impair. Le théorème 31.32 implique que \mathbf{Z}_n^* est un groupe cyclique : il contient un élément générateur g tel que $\text{ord}_n(g) = |\mathbf{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$. D'après l'équation (31.39), on a $g^{n-1} \equiv 1 \pmod{n}$. Le théorème du logarithme discret (théorème 37.33, en prenant $y = 0$, implique alors que $n - 1 \equiv 0 \pmod{\phi(n)}$, soit

$$(p - 1)p^{e-1} \mid p^e - 1.$$

C'est une contradiction pour $e > 1$, car $(p - 1)p^{e-1}$ est divisible par le nombre premier p mais $p^e - 1$ ne l'est pas. n n'est donc pas une puissance d'un nombre premier.

Comme le nombre composé impair n n'est pas une puissance d'un nombre premier, on peut le décomposer en un produit $n_1 n_2$, où n_1 et n_2 sont des nombres impairs plus grands que 1 et premiers entre eux. (Il existe plusieurs moyens d'y parvenir, et peu importe celui qui est choisi. Par exemple, si $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, alors on peut prendre $n_1 = p_1^{e_1}$ et $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Rappelons-nous que l'on définit t et u de telle sorte que $n - 1 = 2^t u$ où $t \geq 1$ et u impair, et que pour une entrée a , TÉMOIN calcule la séquence

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \dots, a^{2^t u} \rangle$$

(tous les calculs sont faits modulo n).

Définissons une paire (v, j) d'entiers comme étant **acceptable** si $v \in \mathbf{Z}_n^*$, $j \in \{0, 1, \dots, t\}$, et

$$v^{2^j u} \equiv -1 \pmod{n}.$$

Il existe certainement des paires acceptables, vu que u est impair ; on peut choisir $v = n - 1$ et $j = 0$, de sorte que $(n - 1, 0)$ est une paire acceptable. Choisissons maintenant le plus grand j possible tel qu'il existe une paire acceptable (v, j) , et fixons v pour que (v, j) soit une paire acceptable. Soit

$$B = \{x \in \mathbf{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Comme B est fermé pour la multiplication modulo n , c'est un sous-groupe de \mathbf{Z}_n^* . D'après le corollaire 31.16, $|B|$ divise donc $|\mathbf{Z}_n^*|$. Chaque non-témoin doit être membre de B , vu que la séquence X produite par un non-témoin doit se composer uniquement de 1 ou alors contenir un -1 pas plus loin que la j ème position, en vertu de la maximalité de j . (Si (a, j') est acceptable, a étant un non-témoin, on doit avoir $j' \leq j$ de par la façon dont on a sélectionné j .)

Nous allons maintenant nous servir de l'existence de v pour démontrer qu'il existe un $w \in \mathbf{Z}_n^* - B$. Comme $v^{2^j u} \equiv -1 \pmod{n}$, on a $v^{2^j u} \equiv -1 \pmod{n_1}$ d'après le corollaire 31.29 du théorème du reste chinois. D'après le corollaire 31.28, il y a un w qui vérifie simultanément les équations

$$\begin{aligned} w &\equiv v \pmod{n_1}, \\ w &\equiv 1 \pmod{n_2}. \end{aligned}$$

Par conséquent,

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1}, \\ w^{2^j u} &\equiv 1 \pmod{n_2}. \end{aligned}$$

D'après le corollaire 31.29, $w^{2^j u} \not\equiv 1 \pmod{n_1}$ implique $w^{2^j u} \not\equiv 1 \pmod{n}$ et $w^{2^j u} \not\equiv -1 \pmod{n_2}$ implique $w^{2^j u} \not\equiv -1 \pmod{n}$. Donc, $w^{2^j u} \not\equiv \pm 1 \pmod{n}$ et donc $w \notin B$.

Reste à montrer que $w \in \mathbf{Z}_n^*$, ce que nous allons faire en commençant par travailler séparément modulo n_1 et modulo n_2 . En travaillant modulo n_1 , on observe que, puisque $v \in \mathbf{Z}_n^*$, on a $\text{pgcd}(v, n) = 1$, et donc on a aussi $\text{pgcd}(v, n_1) = 1$; si v n'a pas de diviseurs communs avec n , c'est qu'il n'a certainement pas de diviseurs communs avec n_1 . Comme $w \equiv v \pmod{n_1}$, on voit que $\text{pgcd}(w, n_1) = 1$. En travaillant modulo n_2 , on observe que $w \equiv 1 \pmod{n_2}$ implique $\text{pgcd}(w, n_2) = 1$. Pour combiner ces résultats, on utilise le théorème 31.6, qui implique que $\text{pgcd}(w, n_1 n_2) = \text{pgcd}(w, n) = 1$. C'est à dire, $w \in \mathbf{Z}_n^*$.

Donc $w \in \mathbf{Z}_n^* - B$, et l'on finit le cas 2 en concluant que B est un sous-groupe propre de \mathbf{Z}_n^* .

Dans l'un ou l'autre cas, on voit que le nombre de témoins du caractère composé de n est au moins $(n - 1)/2$. \square

Théorème 31.39 *Pour tout entier impair $n > 2$ et pour tout entier positif s , la probabilité que MILLER-RABIN(n, s) se trompe est au plus égale à 2^{-s} .*

Démonstration : En utilisant le théorème 31.38, on voit que, si n est composé, chaque exécution de la boucle **pour** des lignes 1–4 a une probabilité d'au moins $1/2$ de découvrir un témoin x du caractère composé de n . MILLER-RABIN ne fait d'erreur que si elle a la malchance de ne pas découvrir de témoin dans chacune des s itérations de la boucle principale. La probabilité d'une telle chaîne d'erreurs est au plus égale à 2^{-s} . \square

Donc, prendre $s = 50$ devrait suffire pour la quasi totalité des applications possibles et imaginables. Si l'on essaie de trouver de grands nombres premiers en appliquant MILLER-RABIN à de grands entiers choisis aléatoirement, alors on peut montrer (ce

que nous ne ferons pas ici) que le fait de choisir une petite valeur de s (par exemple, 3) offre très peu de risques de conduire à des résultats erronés. Autrement dit : pour un entier composé impair n choisi aléatoirement, le nombre attendu de non-témoins du caractère composé de n a de fortes chances d'être très inférieur à $(n - 1)/2$. Si l'entier n n'est pas choisi aléatoirement, cependant, le mieux que l'on puisse prouver est que le nombre de non-témoins est au plus égal à $(n - 1)/4$, et ce en utilisant une version améliorée du théorème 31.38. Qui plus est, il existe effectivement des entiers n pour lesquels le nombre de non-témoins est $(n - 1)/4$.

Exercices

31.8.1 Démontrer que, si un entier impair $n > 1$ n'est ni premier ni puissance d'un nombre premier, alors il existe une racine carrée non triviale de 1 modulo n .

31.8.2 * Il est possible de renforcer légèrement le théorème d'Euler, pour lui donner la forme

$$a^{\lambda(n)} \equiv 1 \pmod{n} \text{ pour tout } a \in \mathbf{Z}_n^*,$$

où $n = p_1^{e_1} \cdots p_r^{e_r}$ et $\lambda(n)$ est défini par

$$\lambda(n) = \text{ppcm}(\phi(p_1^{e_1}), \dots, \phi(p_r^{e_r})). \quad (31.40)$$

Démontrer que $\lambda(n) \mid \phi(n)$. Un nombre composé n est un nombre de Carmichael si $\lambda(n) \mid n - 1$. Le plus petit nombre de Carmichael est $561 = 3 \cdot 11 \cdot 17$; ici, $\lambda(n) = \text{ppcm}(2, 10, 16) = 80$, qui divise 560. Démontrer que les nombres de Carmichael doivent être à la fois « dépourvus de carrés » (non divisibles par le carré d'un nombre premier) et produits d'au moins trois nombres premiers. C'est pour cette raison qu'on ne les rencontre pas très souvent.

31.8.3 Démontrer que, si x est une racine carrée non triviale de 1 modulo n , alors $\text{pgcd}(x - 1, n)$ et $\text{pgcd}(x + 1, n)$ sont tous les deux des diviseurs non triviaux de n .

31.9* FACTORISATION DES ENTIERS

Supposons qu'on se donne un entier n à *factoriser*, c'est-à-dire à décomposer en un produit de nombres premiers. Le test de primarité de la section précédente nous dirait que n est composé, mais ne dirait pas quels sont les facteurs premiers de n . Factoriser un grand entier n semble beaucoup plus difficile que de simplement déterminer s'il est premier ou composé. À ce jour, même avec les ordinateurs les plus puissants et les algorithmes les plus élaborés, on est incapable de factoriser un nombre arbitraire de 1024 bits.

a) Heuristique rho de Pollard

Les essais de division par tous les entiers inférieurs à B permettent de factoriser entièrement tout nombre inférieur à B^2 . Pour la même quantité de travail, la procédure suivante saura factoriser n'importe quel nombre inférieur à B^4 (à moins d'être

malchanceux). La procédure étant simplement une heuristique, son succès n'est pas garanti, pas plus que son temps d'exécution ; cela dit, elle s'avère très efficace en pratique. Autre avantage de POLLARD-RHO : elle n'utilise qu'un nombre constant d'emplacements mémoire. (Il est facile d'implémenter Pollard-Rho sur une calculatrice programmable pour factoriser de petits nombres.)

POLLARD-RHO(n)

```

1   $i \leftarrow 1$ 
2   $x_1 \leftarrow \text{RANDOM}(0, n - 1)$ 
3   $y \leftarrow x_1$ 
4   $k \leftarrow 2$ 
5  tant que VRAI
6    faire  $i \leftarrow i + 1$ 
7       $x_i \leftarrow (x_{i-1}^2 - 1) \bmod n$ 
8       $d \leftarrow \text{pgcd}(y - x_i, n)$ 
9      si  $d \neq 1$  et  $d \neq n$ 
10     alors afficher  $d$ 
11     si  $i = k$ 
12     alors  $y \leftarrow x_i$ 
13      $k \leftarrow 2k$ 
```

La procédure fonctionne de la manière suivante. Les lignes 1–2 initialisent i à 1 et x_1 à une valeur choisie aléatoirement dans \mathbf{Z}_n . La boucle **tant que** commençant à la ligne 5 recherche sans s'arrêter les facteurs de n . Durant chaque itération de la boucle **tant que**, la récurrence

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \quad (31.41)$$

est utilisée à la ligne 7 pour produire la prochaine valeur de x_i dans la séquence infinie

$$x_1, x_2, x_3, x_4, \dots ; \quad (31.42)$$

la valeur de i correspondante est incrémentée à la ligne 6. Pour des raisons de clarté, le code est écrit à l'aide de variables x_i indicées, mais le programme fonctionne de la même manière en supprimant tous les indices, puisque seule la valeur de x_i la plus récente a besoin d'être conservée. Avec cette modification, la procédure ne consomme qu'un nombre constant d'emplacements mémoire.

De temps à autre, le programme mémorise la valeur de x_i la plus récemment générée dans la variable y . Plus précisément, les valeurs mémorisées sont celles dont les indices sont des puissances de 2 :

$$x_1, x_2, x_4, x_8, x_{16}, \dots$$

La ligne 3 sauvegarde la valeur x_1 et la ligne 12 sauvegarde x_k chaque fois que i est égal à k . La variable k est initialisée à 2 à la ligne 4, et k est doublé à la ligne 13 à chaque mise à jour de y . La variable k suit donc la séquence 1, 2, 4, 8, ... et donne toujours l'indice de la prochaine valeur x_k à sauver dans y .

Les lignes 8–10 essaient de trouver un facteur de n , à l'aide de la valeur sauvegardée y et de la valeur courante de x_i . En particulier, la ligne 8 calcule le plus grand commun diviseur $d = \text{pgcd}(y - x_i, n)$. Si d est un diviseur non trivial de n (testé en ligne 9), alors la ligne 10 affiche d .

Cette procédure peut paraître quelque peu étrange au premier abord. Notez, cependant, que POLLARD-RHO n'imprime jamais de réponse incorrecte ; tout nombre imprimé est un diviseur non trivial de n . Toutefois, POLLARD-RHO peut très bien ne rien imprimer du tout ; il n'existe aucune garantie qu'un résultat quelconque soit produit. Nous verrons cependant qu'il y a de bonnes raisons d'espérer que POLLARD-RHO imprime un facteur p de n après $\Theta(\sqrt{p})$ itérations de la boucle **tant que**. Donc, si n est composé, on peut s'attendre à ce que cette procédure découvre assez de diviseurs pour factoriser entièrement n après environ $n^{1/4}$ mises à jour, puisque tout facteur premier p de n , sauf peut-être le plus grand, est inférieur à \sqrt{n} .

Nous commencerons l'analyse du comportement de cette procédure en étudiant le temps que met une séquence aléatoire modulo n pour répéter une valeur. Puisque \mathbf{Z}_n est fini et que chaque valeur de la séquence (31.42) ne dépend que de la valeur précédente, la séquence (31.42) finira inévitablement par se répéter. Une fois atteint un x_i tel que $x_i = x_j$ pour un certain $j < i$, on se trouve à l'intérieur d'un cycle, puisque $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$, etc. Le nom « heuristique rho » vient du fait, comme le montre la figure 31.7, qu'on peut représenter la séquence x_1, x_2, \dots, x_{j-1} par la « queue » de la lettre rho, et le cycle x_j, x_{j+1}, \dots, x_i par le « corps » du rho.

Intéressons-nous au temps pris par la séquence pour commencer à se répéter. Ce n'est pas exactement ce dont nous avons besoin, mais nous verrons ensuite comment modifier la démonstration.

Supposons, dans le cadre de cette estimation, que la fonction $(x^2 - 1) \bmod n$ se comporte comme une fonction « aléatoire ». Bien sûr, elle n'est pas vraiment aléatoire, mais cette hypothèse aboutit à des résultats qui sont cohérents avec le comportement observé de POLLARD-RHO. On peut alors considérer chaque x_i comme étant choisi indépendamment dans \mathbf{Z}_n , selon une distribution uniforme sur \mathbf{Z}_n . D'après l'analyse du paradoxe de l'anniversaire de la section 5.4.1, le nombre moyen d'étapes effectuées avant que la séquence commence à boucler est $\Theta(\sqrt{n})$.

Voyons maintenant la modification nécessaire. Soit p un facteur non trivial de n tel que $\text{pgcd}(p, n/p) = 1$. Par exemple, si n a pour factorisation $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, on pourra donner à p la valeur $p_1^{e_1}$. (Si $e_1 = 1$, alors p est tout simplement le plus petit facteur premier de n , un bon exemple à garder à l'esprit.) La séquence $\langle x_i \rangle$ induit une séquence correspondante $\langle x'_i \rangle$ modulo p , où

$$x'_i = x_i \bmod p$$

pour tout i .

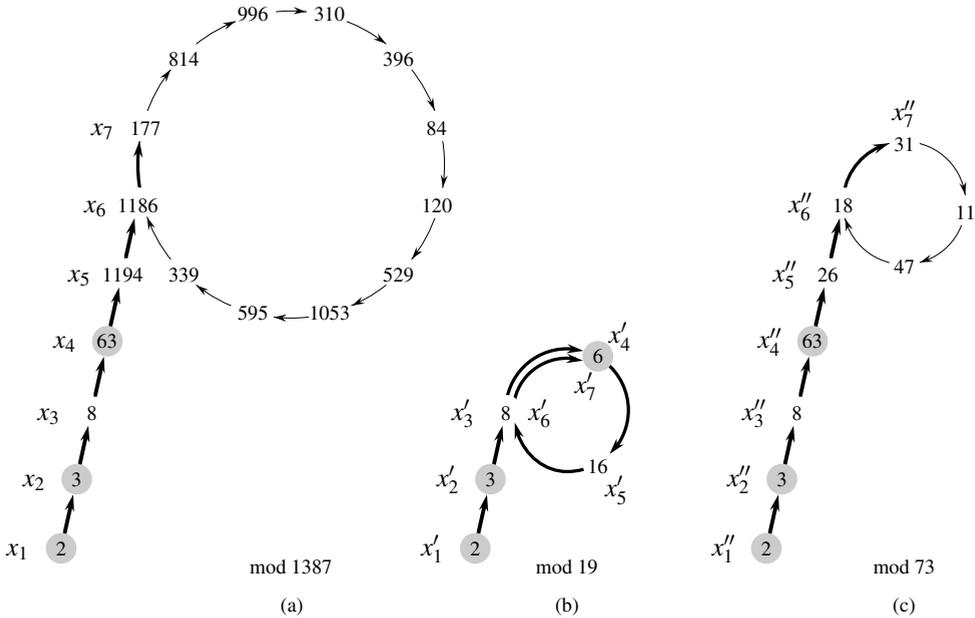


Figure 31.7 Heuristique rho de Pollard. (a) Les valeurs produites par la récurrence $x_{i+1} \leftarrow (x_i^2 - 1) \pmod{1387}$, à partir de $x_1 = 2$. La décomposition en facteurs premiers de 1387 est $19 \cdot 73$. Les flèches épaisses indiquent les étapes d'itérations exécutées avant la découverte du facteur 19. Les flèches fines pointent vers les valeurs non atteintes dans l'itération, pour illustrer la silhouette « rho ». Les valeurs en gris sont les y stockés par POLLARD-RHO. Le facteur 19 est découvert après que $x_7 = 177$ est atteint, quand $\text{pgcd}(63 - 177, 1387) = 19$ est calculé. La première valeur x qui serait répétée est 1186, mais le facteur 19 est découvert avant que cette valeur soit atteinte. (b) Les valeurs produites par la même récurrence, modulo 19. Toutes les valeurs x_i données à la partie (a) sont équivalentes, modulo 19, aux valeurs x'_i montrées ici. Par exemple, $x_4 = 63$ et $x_7 = 177$ sont tous les deux équivalents à 6, modulo 19. (c) Les valeurs produites par la même récurrence, modulo 73. Toutes les valeurs x_i données à la partie (a) sont équivalentes, modulo 73, aux valeurs x''_i montrées ici. D'après le théorème du reste chinois, chaque nœud de la partie (a) correspond à une paire de nœuds, l'un de la partie (b) et l'autre de la partie (c).

En outre, comme f_n n'est définie qu'à l'aide d'opérations arithmétiques (élévation au carré et soustraction) modulo n , on voit que l'on peut calculer x'_{i+1} à partir de x'_i ; la vue « modulo p » de la séquence est une version réduite de ce qui se passe modulo n :

$$\begin{aligned}
 x'_{i+1} &= x_{i+1} \pmod{p} \\
 &= f_n(x_i) \pmod{p} \\
 &= ((x_i^2 - 1) \pmod{n}) \pmod{p} \\
 &= (x_i^2 - 1) \pmod{p} && \text{(d'après exercice 31.1.6)} \\
 &= ((x_i \pmod{p})^2 - 1) \pmod{p} \\
 &= ((x'_i)^2 - 1) \pmod{p} \\
 &= f_p(x'_i).
 \end{aligned}$$

Donc, même si l'on ne calcule pas explicitement la séquence $\langle x'_i \rangle$, cette séquence est bien définie et obéit à la même récurrence que la séquence $\langle x_i \rangle$.

En raisonnant comme précédemment, on trouve que le nombre moyen d'étapes avant répétition de la séquence $\langle x'_i \rangle$ est $\Theta(\sqrt{p})$. Si p est petit comparé à n , la séquence $\langle x'_i \rangle$ risque de réapparaître beaucoup plus vite que la séquence $\langle x_i \rangle$. En effet, la séquence $\langle x'_i \rangle$ se répète dès que deux éléments de la séquence $\langle x_i \rangle$ sont équivalents modulo p (au lieu de modulo n). Cet état de fait est illustré sur la figure 31.7, aux parties (b) et (c).

Soit t l'indice de la première valeur répétée dans la séquence $\langle x'_i \rangle$, et soit $u > 0$ la longueur du cycle produit à cette occasion. En d'autres termes, t et $u > 0$ sont les plus petites valeurs telles que $x'_{t+i} = x'_{t+u+i}$ pour tout $i \geq 0$. D'après les raisonnements précédents, les valeurs attendues de t et u sont toutes les deux $\Theta(\sqrt{p})$. Notez que, si $x'_{t+i} = x'_{t+u+i}$, alors $p \mid (x_{t+u+i} - x_{t+i})$. Donc, $\text{pgcd}(x_{t+u+i} - x_{t+i}, n) > 1$.

De ce fait, une fois que POLLARD-RHO a sauvegardé dans y une valeur x_k telle que $k \geq t$, alors $y \bmod p$ est toujours sur le cycle modulo p . (Si une nouvelle valeur est sauvegardée dans y , cette valeur se trouve également sur le cycle modulo p .) À un certain moment, k finit par prendre une valeur supérieure à u , et la procédure fait alors une boucle complète autour du cycle modulo p sans changer la valeur de y . Un facteur de n est ensuite découvert quand x_i « repasse sur » la valeur de y précédemment stockée, modulo p , c'est-à-dire quand $x_i \equiv y \pmod{p}$.

On peut supposer que le facteur trouvé est le facteur p , bien qu'il puisse aussi être un multiple de p . Comme les valeurs attendues de t et de u sont toutes les deux égales à $\Theta(\sqrt{p})$, le nombre d'étapes requises pour produire le facteur p est $\Theta(\sqrt{p})$.

Cet algorithme peut ne pas se comporter comme prévu, pour deux raisons. Primo, l'analyse heuristique du temps d'exécution n'est pas rigoureuse, et il est possible que le cycle de valeurs, modulo p , ait une taille beaucoup plus grande à \sqrt{p} . Dans ce cas, l'algorithme fonctionne parfaitement, mais s'exécute beaucoup plus lentement que souhaité. En pratique, on voit très rarement cette situation. Secundo, les diviseurs de n produits par cet algorithme pourraient toujours être l'un des deux facteurs triviaux 1 ou n . Par exemple, supposons que $n = pq$, où p et q sont premiers. Il peut arriver que les valeurs de t et de u pour p soient identiques aux valeurs de t et de u pour q , et ainsi que le facteur p soit toujours découvert au cours de la même opération pgcd que celle révélant le facteur q . Les deux facteurs étant révélés en même temps, le facteur trivial $pq = n$ l'est du même coup, ce qui s'avère inutile. Là encore, il semble que ce ne soit pas un vrai problème en pratique. Si nécessaire, l'heuristique peut être réinitialisée avec une récurrence différente de la forme $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$. (On évitera les valeurs $c = 0$ et $c = 2$ pour des raisons que nous n'explorerons pas ici, mais d'autres valeurs fonctionneront correctement.)

Bien entendu, cette analyse est heuristique et n'est pas rigoureuse, puisque la récurrence n'est pas vraiment « aléatoire ». Néanmoins, la procédure fonctionne bien en pratique, et son efficacité semble correspondre à celle prévue par cette analyse

heuristique. C'est la méthode à privilégier pour trouver les petits facteurs premiers d'un grand entier. Pour factoriser entièrement un nombre composé n de β bits, il suffit de trouver tous les facteurs premiers inférieurs à $\lfloor n^{1/2} \rfloor$, et on peut donc s'attendre à ce que POLLARD-RHO nécessite au plus $n^{1/4} = 2^{\beta/4}$ opérations arithmétiques et au plus $n^{1/4}\beta^3 = 2^{\beta/4}\beta^3$ opérations de bits. La capacité de POLLARD-RHO à trouver un petit facteur p de n avec un nombre attendu $\Theta(\sqrt{p})$ d'opérations arithmétiques est souvent son avantage le plus séduisant.

Exercices

31.9.1 Si l'on regarde l'historique de l'exécution montré à la figure 31.7(a), à quel moment POLLARD-RHO imprime-t-elle le facteur 73 de 1387 ?

31.9.2 Supposons qu'on se donne une fonction $f : \mathbf{Z}_n \rightarrow \mathbf{Z}_n$ et une valeur initiale $x_0 \in \mathbf{Z}_n$. On définit $x_i = f(x_{i-1})$ pour $i = 1, 2, \dots$. Soient t et $u > 0$ les deux plus petites valeurs telles que $x_{t+i} = x_{t+u+i}$ pour $i = 0, 1, \dots$. Dans la terminologie de l'algorithme rho de Pollard, t est la longueur de la queue et u est la longueur du cycle du rho. Donner un algorithme efficace permettant de déterminer t et u exactement, et analyser son temps d'exécution.

31.9.3 En combien d'étapes peut-on s'attendre à ce que POLLARD-RHO découvre un facteur de la forme p^e , où p est premier et $e > 1$?

31.9.4 * Un inconvénient de POLLARD-RHO, telle qu'elle est écrite ici, est qu'elle demande un calcul de pgcd à chaque étape de la récurrence. D'aucuns ont suggéré de regrouper les calculs de pgcd, et ce en accumulant le produit de plusieurs x_i consécutifs puis en utilisant ce produit au lieu de x_i dans le calcul de pgcd. Décrire soigneusement la manière d'implémenter cette idée, la justifier, et donner la taille de regroupement la plus efficace quand on travaille sur un nombre n de β bits.

PROBLÈMES

31.1. Algorithme du pgcd binaire

Sur la plupart des ordinateurs, les opérations de soustraction, de test de la parité (paire ou impaire) d'un entier binaire, et de division par deux, peuvent être effectuées plus rapidement que le calcul des restes. Ce problème s'intéresse à l'*algorithme du pgcd binaire*, qui évite les calculs de reste utilisés dans l'algorithme d'Euclide.

- Démontrer que si a et b sont tous les deux pairs, alors $\text{pgcd}(a, b) = 2 \text{pgcd}(a/2, b/2)$.
- Démontrer que si a est impair et b pair, alors $\text{pgcd}(a, b) = \text{pgcd}(a, b/2)$.
- Démontrer que si a et b sont tous deux impairs, $\text{pgcd}(a, b) = \text{pgcd}((a - b)/2, b)$.

- d. Mettre au point un algorithme de pgcd binaire efficace prenant en entrée deux entiers a et b , où $a \geq b$, et s'exécutant en temps $O(\lg a)$. On suppose que chaque soustraction, test de parité, et division par deux, peut s'effectuer en temps unitaire.

31.2. Analyse des opérations de bits dans l'algorithme d'Euclide

- a. On considère l'algorithme habituel « sur papier » utilisé pour les divisions longues : division de a par b , ce qui donne un quotient q et un reste r . Montrer que cette méthode nécessite $O((1 + \lg q) \lg b)$ opérations de bits.
- b. On définit $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Montrer que le nombre d'opérations de bits effectuées par EUCLIDE en réduisant le calcul de $\text{pgcd}(a, b)$ à celui de $\text{pgcd}(b, a \bmod b)$ vaut au plus $c(\mu(a, b) - \mu(b, a \bmod b))$ pour une certaine constante $c > 0$ suffisamment grande.
- c. Montrer que EUCLIDE(a, b) requiert de manière générale $O(\mu(a, b))$ opérations de bits, et $O(\beta^2)$ opérations de bits quand on l'applique à deux entrées de β bits.

31.3. Trois algorithmes pour les nombres de Fibonacci

Ce problème compare l'efficacité de trois méthodes de calcul du n ième nombre de Fibonacci F_n , sachant n . On suppose que le coût de l'addition, de la soustraction, ou de la multiplication de deux nombres est $O(1)$, quelle que soit la taille des nombres en question.

- a. Montrer que le temps d'exécution de la méthode récursive naturelle pour calculer F_n , basée sur la récurrence (3.21) est exponentiel en n .
- b. Montrer comment calculer F_n en temps $O(n)$ en utilisant le recensement.
- c. Montrer comment calculer F_n en temps $O(\lg n)$ en ne faisant appel qu'à l'addition et à la multiplication des entiers. (*Conseil* : Considérer la matrice

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

et ses puissances.)

- d. Supposons à présent que l'addition de deux nombres de β bits prenne un temps $\Theta(\beta)$ et que leur multiplication prenne $\Theta(\beta^2)$. Quel est le temps d'exécution de ces trois méthodes avec cette mesure plus raisonnable du coût des opérations arithmétiques élémentaires ?

31.4. Résidus quadratiques

Soit p un nombre premier impair. Un nombre $a \in \mathbb{Z}_p^*$ est un *résidu quadratique* si l'équation $x^2 = a \pmod{p}$ a une solution pour l'inconnue x .

- a. Montrer qu'il existe exactement $(p - 1)/2$ résidus quadratiques modulo p .

- b. Si p est premier, on définit le **symbole de Legendre** $\left(\frac{a}{p}\right)$, pour $a \in \mathbf{Z}_p^*$, par 1 si a est un résidu quadratique modulo p et -1 sinon. Prouver que si $a \in \mathbf{Z}_p^*$, alors

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Donner un algorithme efficace permettant de déterminer si un nombre donné a est ou non un résidu quadratique modulo p . Analyser l'efficacité de votre algorithme.

- c. Démontrer que si p est un nombre premier de la forme $4k+3$ et a est un résidu quadratique de \mathbf{Z}_p^* , alors $a^{k+1} \pmod{p}$ est une racine carrée de a , modulo p . Quel temps faudra-t-il pour trouver la racine carrée d'un résidu quadratique a modulo p ?
- d. Décrire un algorithme randomisé efficace permettant de trouver un résidu non quadratique, modulo un nombre premier p arbitraire (c'est-à-dire, un membre de \mathbf{Z}_p^* qui n'est pas un résidu quadratique). Combien d'opérations arithmétiques faudra-t-il à votre algorithme en moyenne ?

NOTES

Niven et Zuckerman [231] offrent une excellente introduction à la théorie des nombres. Knuth [183] propose une bonne étude des algorithmes de recherche du plus grand commun diviseur, ainsi que d'autres algorithmes de base de la théorie des nombres. Riesel [258] et Bach [28] proposent une vue plus moderne de la théorie calculatoire des nombres. Dixon [78] donne une vue d'ensemble de la factorisation et du test de primarité. Les communications publiées par Pomerance [245] comportent plusieurs articles excellents. Plus récemment, Bach et Shallit [29] ont fourni une présentation exceptionnelle des bases de la théorie calculatoire des nombres.

Knuth [183] s'intéresse à l'origine de l'algorithme d'Euclide. Celui-ci apparaît dans le Livre 7, Propositions 1 et 2, des *Eléments* du mathématicien grec Euclide, écrit autour de 300 AV. J.-C. La description d'Euclide pourrait s'être inspirée d'un algorithme dû à Eudoxe aux environs de 375 AV. J.-C. L'algorithme d'Euclide peut s'honorer d'être le plus ancien algorithme non trivial ; il n'est concurrencé que par un algorithme de multiplication qui était connu des anciens Egyptiens. Shallit [274] retrace l'histoire de l'analyse de l'algorithme d'Euclide.

Knuth attribue un cas particulier du théorème du reste chinois (théorème 31.27) au mathématicien chinois Sun-Tsü, qui vécut entre 200 AV. J.-C. et 200 AP. J.-C. (la date est incertaine). Le même cas particulier fut donné par le mathématicien grec Nicomaque autour de 100 AP. J.-C. Il fut généralisé par Chhin Chiu-Shao en 1247. Le théorème du reste chinois fut enfin établi et démontré dans sa pleine généralité par L. Euler en 1734.

L'algorithme du test de primarité randomisé présenté dans ce chapitre est dû à Miller [221] et Rabin [254] ; c'est l'algorithme randomisé le plus rapide à ce jour pour ce problème, à des facteurs constants près. La démonstration du théorème 31.39 est une légère adaptation de celle suggérée par Bach [27]. La démonstration d'un résultat plus fort pour MILLER-RABIN fut donnée par Monier [224, 225]. La randomisation semble être indispensable pour obtenir un algorithme de test de primarité à temps polynomial. L'algorithme déterministe le plus rapide connu à ce jour pour le test de primarité est la version Cohen-Lenstra [65] du test de

primarité publié par Adleman, Pomerance et Rumely [3]. Lorsqu'il teste le caractère premier d'un nombre n de longueur $\lceil \lg(n+1) \rceil$, l'algorithme s'exécute en temps $(\lg n)^{O(\lg \lg \lg n)}$, ce qui est très légèrement supra polynomial.

Le problème consistant à trouver de grands nombres entiers « aléatoires » est joliment décrit dans un article de Beauchemin, Brassard, Crépeau, Goutier et Pomerance [33].

Le concept de cryptosystème à clés publiques est dû à Diffie et Hellman [74]. Le cryptosystème RSA fut proposé en 1977 par Rivest, Shamir et Adleman [259]. Depuis lors, le champ de la cryptographie s'est considérablement enrichi. Notre connaissance du cryptosystème RSA s'est approfondie, et les variantes modernes utilisent des raffinements majeurs des techniques basiques traitées ici. En outre, de nombreuses techniques nouvelles ont été développées pour démontrer la sécurité des cryptosystèmes. Notamment, Goldwasser et Micali [123] montrent que la randomisation peut être un outil efficace dans l'élaboration de modèles faibles de chiffrement à clés publiques. Pour les signatures, Goldwasser, Micali et Rivest [124] présentent un modèle de signature numérique pour lequel on peut démontrer que tout type concevable d'effraction est aussi complexe que la factorisation. Menezes et al. [220] fournissent une présentation de la cryptographie appliquée.

L'heuristique rho pour la factorisation d'un entier a été inventée par Pollard [242]. La version présentée ici est une variante proposée par Brent [48].

Les meilleurs algorithmes de factorisation de grands entiers ont des temps d'exécution qui croissent de façon à peu près exponentielle avec la racine cubique de la longueur du nombre n à factoriser. L'algorithme général de factorisation seive, créé par Buhler et al. [51] en tant qu'extension des concepts de l'algorithme de factorisation seive de Pollard [243] et Lenstra et al. [201], puis amélioré par Coppersmith [69] et d'autres, est peut-être le plus performant des algorithmes de ce genre pour les entrées grandes. Il est difficile de donner une analyse rigoureuse de cet algorithme mais, moyennant des hypothèses raisonnables, on peut estimer son temps d'exécution à $L(1/3, n)^{1.902+o(1)}$, avec $L(\alpha, n) = e^{(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$.

La méthode de la courbe elliptique, due à Lenstra [202], est plus efficace, pour certaines entrées, que la méthode seive ; en effet, à l'instar de la méthode rho de Pollard, elle peut trouver un petit facteur premier p assez rapidement. Avec cette méthode, le temps pour trouver p est estimé à $L(1/2, p)^{\sqrt{2}+o(1)}$.

Chapitre 32

Recherche de chaînes de caractères

Trouver toutes les occurrences d'une chaîne de caractères (motifs ou mots) dans un texte est un problème qu'on rencontre fréquemment dans les éditeurs de texte. Le plus souvent, le texte est un document en cours d'édition et la chaîne de caractères recherchée est un mot particulier fourni par l'utilisateur. Des algorithmes efficaces pour ce problème peuvent grandement améliorer la réactivité du programme d'édition. Les algorithmes de recherche de chaîne de caractères sont également utilisés, par exemple, pour trouver telle ou telle chaîne de caractères dans une séquence ADN.

On formalise le *problème de la recherche de chaîne de caractères* de la manière suivante. On suppose que le texte est un tableau $T[1..n]$ de longueur n et que la chaîne de caractères est un tableau $P[1..m]$ de longueur $m \leq n$. On suppose en outre que les éléments de P et T sont des caractères appartenant à un alphabet fini Σ . Par exemple, on pourra avoir $\Sigma = \{0, 1\}$ ou $\Sigma = \{a, b, \dots, z\}$.

On dit que la chaîne P *apparaît avec un décalage* s dans le texte T (ou, de façon équivalente, que la chaîne P *apparaît à la position* $s + 1$ dans le texte T) si $0 \leq s \leq n - m$ et $T[s + 1..s + m] = P[1..m]$ (c'est-à-dire si $T[s + j] = P[j]$ pour $1 \leq j \leq m$). Si P apparaît avec un décalage s dans T , alors le décalage s est dit *valide* ; sinon, s est dit *invalid*. Le problème de la recherche de chaîne de caractères revient à trouver tous les décalages valides pour lesquels une chaîne P donnée apparaît dans un texte T donné. La figure 32.1 illustre ces définitions.

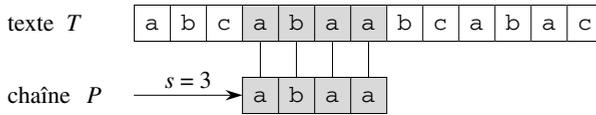


Figure 32.1 Le problème de la recherche de chaîne de caractères. Le but est de trouver toutes les occurrences de la chaîne $P = abaa$ dans le texte $T = abcabaabcabac$. La chaîne de caractères n'apparaît qu'une seule fois, avec le décalage $s = 3$. On dit que le décalage $s = 3$ est valide. Chaque caractère de la chaîne de caractères est relié par une ligne verticale au caractère correspondant dans le texte, et tous les caractères appariés sont sur fond gris.

Exception faite de l'algorithme naïf et bestial que nous reverrons à la section 32.1, chaque algorithme de recherche de motif que nous étudierons dans ce chapitre effectue un certain pré traitement, qui dépend de la chaîne recherchée, puis trouve tous les décalages valides ; cette dernière phase porte le nom de « recherche de correspondance » (matching). La figure 32.2 montre les temps de pré traitement et de recherche de correspondance des divers algorithmes qui seront vus dans ce chapitre. Le temps d'exécution total de chaque algorithme est la somme des temps de pré traitement et de recherche de correspondance. La section 32.2 présentera un algorithme intéressant de recherche de motif, dû à Rabin et à Karp. Le temps d'exécution $\Theta((n - m + 1)m)$, dans le cas le plus défavorable, de cet algorithme n'est pas meilleur que celui de la méthode simpliste, mais il fonctionne mieux en moyenne et en pratique. Sans compter qu'on peut l'étendre sans difficulté à d'autres problèmes de recherche de motif. La section 32.3 décrira ensuite un algorithme de recherche de chaîne qui commence par construire un automate fini spécifiquement conçu pour la recherche dans un texte des occurrences du motif P donné. Cet algorithme prend un temps de pré traitement $O(m|\Sigma|)$, mais ne prend qu'un temps $\Theta(n)$ pour la recherche de correspondance. L'algorithme KMP (Knuth-Morris-Pratt), qui ressemble au précédent mais en plus astucieux, sera traité à la section 32.4 ; l'algorithme KMP a le même temps $\Theta(n)$ pour la recherche de correspondance, mais fait tomber le temps de pré traitement à $\Theta(m)$.

Algorithme	Temps de pré traitement	Temps de recherche des correspondances
naïf	0	$O((n - m + 1)m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Automate fini	$O(m \Sigma)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 32.2 Algorithmes de recherche de chaîne étudiés dans ce chapitre, avec leurs durées de pré traitement et de recherche de correspondance.

a) Notation et terminologie

Nous appellerons Σ^* (lire « sigma-étoile ») l'ensemble de toutes les chaînes de longueur finie utilisant les caractères de l'alphabet Σ . Dans ce chapitre, on ne s'intéressera qu'aux chaînes de longueur finie. La **chaîne vide** de longueur zéro, notée ε , appartient également à Σ^* . La longueur d'une chaîne x est notée $|x|$. La **concaténation** de deux chaînes x et y , notée xy , a pour longueur $|x| + |y|$ et est composée des caractères de x suivis des caractères de y .

On dit qu'une chaîne w est un **préfixe** d'une chaîne x , notée $w \sqsubset x$, si $x = wy$ pour une certaine chaîne $y \in \Sigma^*$. Notez que si $w \sqsubset x$, alors $|w| \leq |x|$. De même, on dira qu'une chaîne w est un **suffixe** d'une chaîne x , notée $w \sqsupset x$, si $x = yw$ pour un certain $y \in \Sigma^*$. On déduit de $w \sqsupset x$ que $|w| \leq |x|$. La chaîne vide ε est à la fois suffixe et préfixe de toute chaîne. Par exemple, on a $ab \sqsubset abcca$ et $cca \sqsupset abcca$. Il est utile de remarquer que, pour toutes chaînes x et y et pour tout caractère a , $x \sqsubset y$ si et seulement si $xa \sqsubset ya$. Notez également que \sqsubset et \sqsupset sont des relations transitives. Le lemme suivant nous sera utile plus tard.

Lemme 32.1 (Lemme de recouvrement des suffixes) *Supposons que x, y et z soient trois chaînes telles que $x \sqsupset z$ et $y \sqsupset z$. Si $|x| \leq |y|$, alors $x \sqsupset y$. Si $|x| \geq |y|$, alors $y \sqsupset x$. Si $|x| = |y|$, alors $x = y$.*

Démonstration : Voir la figure 32.3 pour une démonstration graphique. □

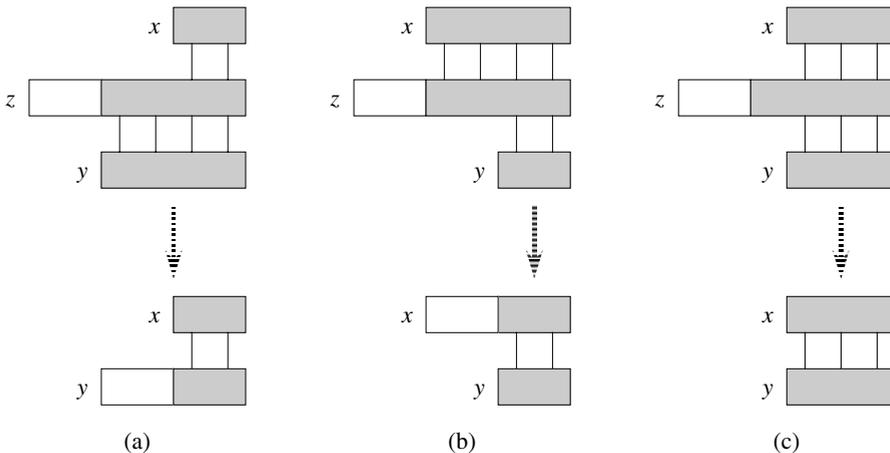


Figure 32.3 Démonstration du lemme 32.1. On suppose que $x \sqsupset z$ et $y \sqsupset z$. Les trois parties de la figure illustrent les trois cas du lemme. Les lignes verticales relient les régions appariées (coloriées en gris) des chaînes. (a) Si $|x| \leq |y|$, alors $x \sqsupset y$. (b) Si $|x| \geq |y|$, alors $y \sqsupset x$. (c) Si $|x| = |y|$, alors $x = y$.

Pour alléger la notation, on notera P_k le préfixe de longueur k $P[1..k]$ de la chaîne de caractères $P[1..m]$. Donc, $P_0 = \varepsilon$ et $P_m = P = P[1..m]$. De même, on notera

T_k le préfixe de longueur k du texte T . Avec cette notation, le problème de la recherche de chaîne de caractères revient à trouver tous les décalages s dans l'intervalle $0 \leq s \leq n - m$ tels que $P \sqsubset T_{s+m}$.

Dans notre pseudo code, on considère la comparaison de deux chaînes de longueurs égales comme une opération primitive. Si les chaînes sont comparées de la gauche vers la droite et que la comparaison s'arrête quand une différence est découverte, on suppose que le temps pris par un tel test est une fonction linéaire du nombre de caractères concordants découverts. Plus précisément, on suppose que le test « $x = y$ » requiert un temps $\Theta(t + 1)$, où t est la longueur de la plus longue chaîne z telle que $z \sqsubset x$ et $z \sqsubset y$. (On écrit $\Theta(t + 1)$ au lieu de $\Theta(t)$ pour gérer le cas où $t = 0$; les premiers caractères comparés ne concordent pas, mais ces comparaisons prennent un certain temps.)

32.1 ALGORITHME NAÏF DE RECHERCHE DE CHAÎNE DE CARACTÈRES

L'algorithme naïf trouve tous les décalages valides en utilisant une boucle qui teste la condition $P[1..m] = T[s+1..s+m]$ pour chacune des $n - m + 1$ valeurs possibles de s .

```

RECHERCHE-NAÏVE( $T, P$ )
1   $n \leftarrow \text{longueur}[T]$ 
2   $m \leftarrow \text{longueur}[P]$ 
3  pour  $s \leftarrow 0$  à  $n - m$ 
4      faire si  $P[1..m] = T[s+1..s+m]$ 
5          alors afficher « La chaîne apparaît avec le décalage »  $s$ 

```

La procédure naïve de recherche de chaîne de caractères peut s'interpréter graphiquement de la façon suivante : on passe sur le texte un « modèle » contenant la chaîne de caractères recherchée et l'on note, au fur et à mesure, les décalages à partir desquels tous les caractères du modèle sont égaux aux caractères correspondants du texte (voir figure 32.4). La boucle **pour** commençant en ligne 3 considère explicitement chaque décalage possible. Le test en ligne 4 détermine si le décalage courant est valide ou non ; ce test suppose une boucle implicite qui teste chaque position de caractère, jusqu'à ce qu'une concordance soit trouvée pour tous les caractères ou qu'une différence soit repérée. La ligne 5 affiche chaque décalage valide s .

La procédure COMPARATEUR-NAÏF prend un temps $O((n - m + 1)m)$, et cette borne est serrée dans le cas le plus défavorable. Par exemple, considérons le texte a^n (une chaîne de n a) et le motif a^m . Pour chacune des $n - m + 1$ valeurs possibles du décalage s , la boucle implicite de la ligne 4 (qui compare les caractères correspondants) s'exécute m fois pour valider le décalage. Le temps d'exécution dans le pire des cas est donc $\Theta((n - m + 1)m)$, ce qui donne $\Theta(n^2)$ si $m = \lfloor n/2 \rfloor$. Le temps d'exécution de

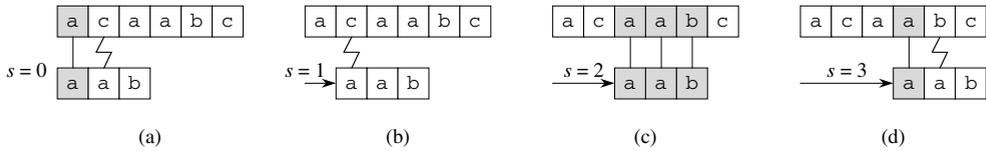


Figure 32.4 L'action du comparateur naïf pour la chaîne $P = aab$ et le texte $T = acaabc$. On peut imaginer le motif P comme un « modèle » qui glisse parallèlement au texte. Les parties (a)–(d) montrent les quatre alignements successifs testés par le comparateur naïf. Dans chaque partie, des lignes verticales relient les régions appariées (en gris) et une ligne brisée relie le premier caractère trouvé qui ne correspond pas (s'il y en a un). Une occurrence de la chaîne de caractères est trouvée, au décalage $s = 2$, illustré en partie (c).

COMPARATEUR-NAÏF est égal au temps de recherche des correspondances, car ici il n'y a point de pré traitement.

Comme nous le verrons, COMPARATEUR-NAÏF n'est pas une procédure optimale pour ce problème. En effet, nous rencontrerons dans ce chapitre un algorithme dont le temps de pré traitement dans le cas le plus défavorable est $\Theta(m)$ et dont le temps de recherche des correspondances, toujours dans le cas le plus défavorable, est $\Theta(n)$. Le comparateur naïf est inefficace parce que les informations obtenues sur le texte pour une valeur de s ne sont pas prises en compte quand on examine les valeurs suivantes de s . Pourtant, cette information pourrait être très précieuse. Par exemple, si $P = aaab$ et si on trouve que $s = 0$ est valide, on peut deviner qu'aucun des décalages 1, 2 et 3 ne sera valide puisque $T[4] = b$. Dans les sections suivantes, on examinera plusieurs moyens d'utiliser efficacement ce type d'information.

Exercices

32.1.1 Donner les comparaisons effectuées par le comparateur naïf pour la chaîne $P = 0001$ dans le texte $T = 000010001010001$.

32.1.2 On suppose que tous les caractères de la chaîne P sont différents. Montrer comment accélérer COMPARATEUR-NAÏF, pour qu'il prenne un temps $O(n)$ sur un texte T de n caractères.

32.1.3 Supposons que la chaîne P et le texte T soient des chaînes de longueurs respectives m et n , dont les caractères sont choisis *aléatoirement* dans l'alphabet à d caractères $\Sigma_d = \{0, 1, \dots, d-1\}$, avec $d \geq 2$. Montrer que le nombre *moyen* de comparaisons entre caractères effectuées par la boucle implicite de la ligne 4 de l'algorithme naïf est

$$(n - m + 1) \frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$$

sur l'ensemble des exécutions de cette boucle. (On suppose que l'algorithme naïf arrête les comparaisons pour un décalage donné dès que deux caractères ne correspondent plus ou dès qu'il a trouvé une correspondance pour la chaîne de caractères au complet.) Pour des chaînes choisies aléatoirement, l'algorithme naïf est donc plutôt efficace.

32.1.4 Supposons que la chaîne P contienne des occurrences d'un *caractère joker* \diamond pouvant remplacer une chaîne de caractères *arbitraire* (même de longueur nulle). Par exemple, la chaîne de caractères $ab\diamond ba\diamond c$ apparaît dans le texte $cabccbacbacab$ sous les formes

$$\begin{array}{ccccccccc} c & ab & cc & ba & cba & c & ab \\ & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \\ & ab & \diamond & ba & \diamond & c & \end{array}$$

et

$$\begin{array}{ccccccccc} c & ab & ccbac & ba & & c & ab \\ & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \underbrace{\hspace{1em}} & \\ & ab & \diamond & ba & \diamond & c & \end{array}$$

Notez que le joker peut apparaître un nombre de fois quelconque dans le motif recherché, mais que l'on suppose qu'il n'apparaît pas dans le texte. Donner un algorithme à temps polynomial qui détermine si une telle chaîne P apparaît dans un texte T donné, puis analyser son temps d'exécution.

32.2 ALGORITHME DE RABIN-KARP

Rabin et Karp ont proposé un algorithme de recherche de chaîne qui fonctionne bien en pratique et qui peut aussi se généraliser à des problèmes voisins, telle la recherche de motifs bidimensionnels. L'algorithme de Rabin-Karp consomme un temps de pré traitement $\Theta(m)$, et son temps d'exécution est $\Theta((n - m + 1)m)$ dans le cas le plus défavorable. Moyennant certaines hypothèses, toutefois, le temps d'exécution moyen est meilleur.

Cet algorithme utilise des notions de la théorie élémentaire des nombres, comme l'équivalence de deux nombres modulo un troisième. On pourra se référer à la section 31.1 pour consulter les définitions.

Pour les besoins de l'exposé, supposons que $\Sigma = \{0, 1, 2, \dots, 9\}$, de sorte que chaque caractère est un chiffre décimal. (Dans le cas général, on peut supposer que chaque caractère est un chiffre exprimé en base d , où $d = |\Sigma|$.) On peut alors voir une chaîne de k caractères consécutifs comme représentant un nombre décimal de longueur k . La chaîne de caractères 31415 correspond donc au nombre décimal 31 415. Étant donnée l'interprétation duale des caractères d'entrée, à la fois comme symboles graphiques et comme chiffres, nous avons trouvé plus commode de les représenter dans cette section comme s'il s'agissait de chiffres, dans notre police de caractères habituelle.

Étant donné une chaîne de caractères $P[1..m]$, on note p sa valeur décimale correspondante. De même, étant donné un texte $T[1..n]$, on note t_s la valeur décimale de la sous-chaîne de longueur m $T[s + 1..s + m]$, pour $s = 0, 1, \dots, n - m$. Bien sûr, $t_s = p$ si et seulement si $T[s + 1..s + m] = P[1..m]$; donc, s est un décalage valide si et seulement si $t_s = p$. Si l'on pouvait calculer p en temps $\Theta(m)$ et toutes les valeurs t_s en un total de $\Theta(n - m + 1)$ ⁽¹⁾, alors on pourrait déterminer tous les décalages s

(1) Nous écrivons $\Theta(n - m + 1)$ au lieu de $\Theta(n - m)$, car il y a $n - m + 1$ valeurs différentes possibles pour s . Le « +1 » est significatif pris dans un sens asymptotique vu que, quand $m = n$, calculer la seule valeur t_s prend un temps $\Theta(1)$ et non un temps $\Theta(0)$.

valides en temps $\Theta(m) + \Theta(n - m + 1) = \Theta(n)$ en comparant p à chacun des t_s . (Pour le moment, nous ne prenons pas en compte la possibilité que p et les t_s soient des nombres très grands.)

On peut calculer p en temps $\Theta(m)$ à l'aide de la règle de Horner (voir section 30.1) :

$$p = P[m] + 10(P[m - 1] + 10(P[m - 2] + \dots + 10(P[2] + 10P[1]) \dots)) .$$

La valeur t_0 peut être calculée de la même manière à partir de $T[1 \dots m]$ en temps $\Theta(m)$.

Pour calculer les valeurs restantes t_1, t_2, \dots, t_{n-m} en temps $\Theta(n - m)$, il suffit d'observer que t_{s+1} peut être calculé à partir de t_s en temps constant, puisque

$$t_{s+1} = 10(t_s - 10^{m-1}T[s + 1]) + T[s + m + 1] . \quad (32.1)$$

Par exemple, si $m = 5$ et $t_s = 31415$, on souhaite supprimer le chiffre le plus significatif $T[s + 1] = 3$ et faire entrer le nouveau chiffre d'ordre inférieur (disons qu'il s'agit de $T[s + 5 + 1] = 2$) pour obtenir

$$t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152 .$$

Soustraire $10^{m-1}T[s + 1]$ a pour effet de supprimer le chiffre d'ordre supérieur de t_s ; multiplier le résultat par 10 décale le nombre d'une position vers la gauche ; ajouter $T[s + m + 1]$ fait apparaître le chiffre d'ordre inférieur voulu. Si la constante 10^{m-1} est pré calculée (ce qu'on peut faire en temps $O(\lg m)$ à l'aide des techniques de la section 31.6 bien que, pour cette application, une méthode plus directe en $O(m)$ soit parfaitement adaptée), alors chaque exécution de l'équation (32.1) prend un nombre constant d'opérations arithmétiques. Donc, on peut calculer p en temps $\Theta(m)$ et t_0, t_1, \dots, t_{n-m} en temps $\Theta(n - m + 1)$, et on peut trouver toutes les occurrences de la chaîne $P[1 \dots m]$ dans le texte $T[1 \dots n]$ avec un temps de pré traitement $\Theta(m)$ et un temps de recherche de correspondances $\Theta(n - m + 1)$.

La seule difficulté de cette procédure est que p et t_s peuvent être trop grands pour que le calcul soit commode. Si P contient m caractères, alors supposer que chaque opération arithmétique sur p (dont la longueur est de m chiffres) prend un « temps constant » n'est plus raisonnable. Heureusement, il existe un remède simple pour ce problème, comme le montre la figure 32.5 : calculer p et les t_s modulo un entier q approprié. Comme les calculs de p , t_0 et de la récurrence (32.1) peuvent tous être effectués modulo q , on voit que p peut être calculé modulo q en $\Theta(m)$ et que tous les t_s peuvent être calculés modulo q en $\Theta(n - m + 1)$. q est choisi le plus souvent comme un entier tel que $10q$ tienne juste dans un seul mot machine, ce qui permet d'effectuer tous les calculs nécessaires avec une arithmétique en simple précision. De manière générale, avec un alphabet d -aire $\{0, 1, \dots, d - 1\}$, on choisit q de telle sorte que dq tienne dans un mot d'ordinateur et on ajuste l'équation de récurrence (32.1) pour qu'elle soit valable modulo q . Elle devient donc

$$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q , \quad (32.2)$$

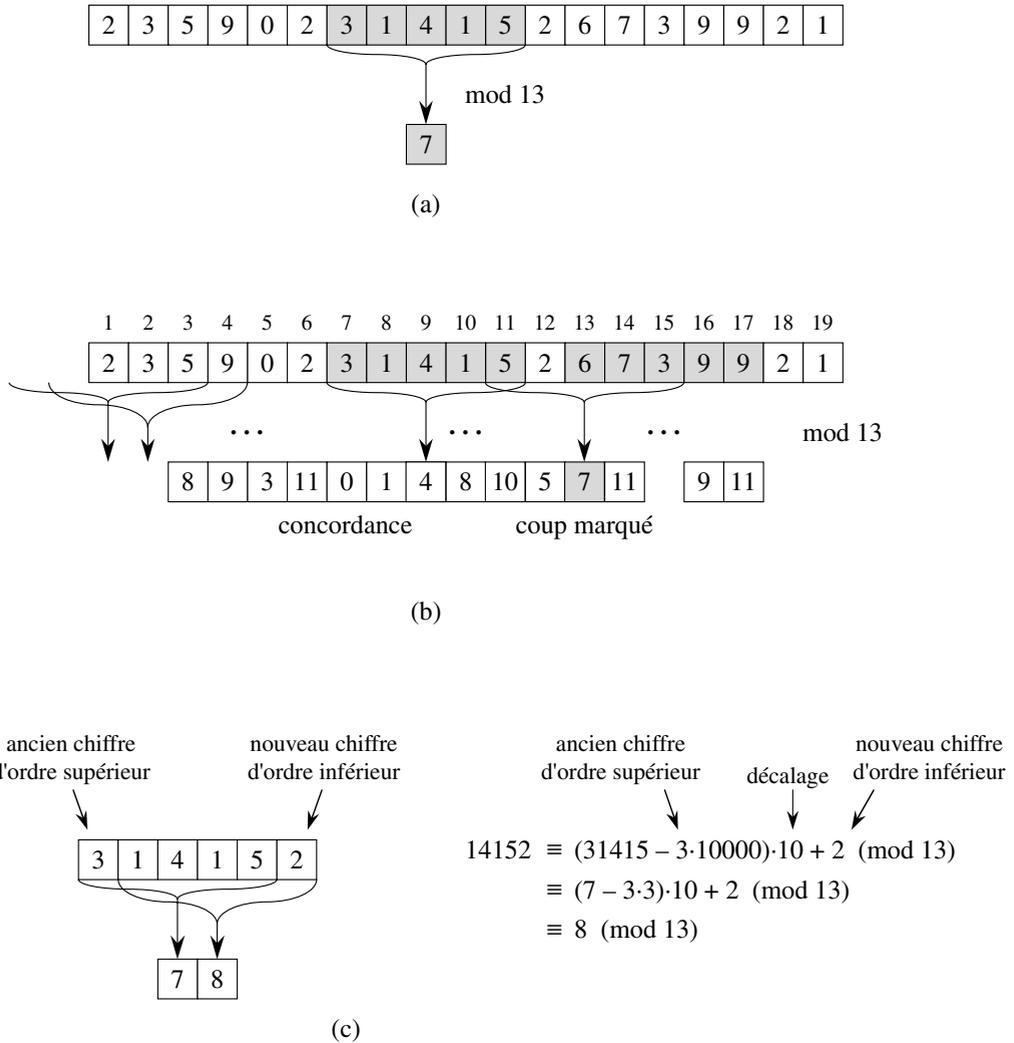


Figure 32.5 L’algorithme de Rabin-Karp. Chaque caractère est un chiffre décimal et on calcule les valeurs modulo 13. **(a)** Une chaîne de texte. Une fenêtre de longueur 5 est représentée en gris. La valeur numérique du nombre en gris est calculée modulo 13 et donne 7. **(b)** La même chaîne avec des valeurs calculées modulo 13 pour chaque position possible d’une fenêtre de longueur 5. En supposant que la valeur de la chaîne de caractères est $P = 31415$, on s’intéresse aux fenêtres dont la valeur modulo 13 est égale à 7, puisque $31415 \equiv 7 \pmod{13}$. Deux fenêtres de ce type sont trouvées (affichées en gris sur la figure). La première, qui commence à la position 7 dans le texte, est bien une occurrence de la chaîne de caractères, tandis que la seconde, qui débute à la position 13 dans le texte, est un coup manqué. **(c)** Le calcul de la valeur d’une fenêtre en temps constant, connaissant la valeur de la fenêtre précédente. La première fenêtre a la valeur 31415. La suppression du chiffre d’ordre supérieur 3, puis le décalage à gauche (en multipliant par 10), et enfin l’addition du chiffre d’ordre inférieur 2, tout cela donne la nouvelle valeur 14152. Toutefois, tous les calculs sont effectués modulo 13, de sorte que la valeur de la première fenêtre est 7 et que celle calculée pour la nouvelle fenêtre est 8.

où $h \equiv d^{m-1} \pmod{q}$ est la valeur du chiffre « 1 » à la position d'ordre supérieur d'une fenêtre de texte à m chiffres.

Toutefois, l'inconvénient de cette technique vient du fait que $t_s \equiv p \pmod{q}$ n'implique pas $t_s = p$. En revanche, si $t_s \not\equiv p \pmod{q}$, on a de manière certaine $t_s \neq p$ et le décalage s est invalide. On peut donc utiliser le test $t_s \equiv p \pmod{q}$ comme test heuristique rapide pour éliminer les décalages invalides. Tout décalage s pour lequel $t_s \equiv p \pmod{q}$ doit être à nouveau testé pour voir si s est bien valide, ou s'il s'agissait d'un **coup manqué**. Ce test peut être fait explicitement en vérifiant la condition $P[1..m] = T[s+1..s+m]$. Si q est assez grand, on peut espérer que ces coups manqués soient assez rares pour que le coût de la vérification supplémentaire soit faible.

La procédure ci-dessous précise ces idées. Les paramètres de la procédure sont le texte T , la chaîne P , la base d (à qui l'on donne le plus souvent la valeur $|\Sigma|$) et le nombre premier q .

```

RABIN-KARP( $T, P, d, q$ )
1   $n \leftarrow \text{longueur}[T]$ 
2   $m \leftarrow \text{longueur}[P]$ 
3   $h \leftarrow d^{m-1} \pmod{q}$ 
4   $p \leftarrow 0$ 
5   $t_0 \leftarrow 0$ 
6  pour  $i \leftarrow 1$  à  $m$                                 ▷ pré traitement
7      faire  $p \leftarrow (dp + P[i]) \pmod{q}$ 
8           $t_0 \leftarrow (dt_0 + T[i]) \pmod{q}$ 
9  pour  $s \leftarrow 0$  à  $n - m$                             ▷ recherche de correspondance
10     faire si  $p = t_s$ 
11         alors si  $P[1..m] = T[s+1..s+m]$ 
12             alors afficher « la chaîne recherchée apparaît
                    à la position »  $s$ 
13         si  $s < n - m$ 
14             alors  $t_{s+1} \leftarrow (d(t_s - T[s+1])h + T[s+m+1]) \pmod{q}$ 

```

La procédure RABIN-KARP fonctionne de la manière suivante. Tous les caractères sont interprétés comme des chiffres en base d . Les indices accompagnant t ne sont là qu'à des fins de clarté ; le programme marche correctement si on laisse tomber les indices. La ligne 3 initialise h en y mettant la valeur qui occupe la position numérique de poids fort d'une fenêtre à m chiffres. Les lignes 4–8 calculent p en tant que valeur de $P[1..m] \pmod{q}$ et t_0 en tant que valeur de $T[1..m] \pmod{q}$. La boucle **pour** des lignes 9–14 examine tous les décalages possibles s , en conservant l'invariant que voici : Pour chaque exécution de la ligne 10, $t_s = T[s+1..s+m] \pmod{q}$.

Si $p = t_s$ en ligne 10 (on a un « coup au but »), alors on regarde si $P[1..m] = T[s+1..s+m]$ en ligne 11 pour éliminer la possibilité qu'il y ait un coup manqué. Tous les décalages valides ayant été détectés sont affichés par la ligne 12. Si $s < n-m$ (testé en ligne 13), alors la boucle **pour** doit être exécutée au moins une fois de plus ; on commence donc par exécuter la ligne 14 pour assurer que l'invariant de boucle restera vrai quand on atteindra derechef la ligne 10. La ligne 14 calcule en temps constant la valeur de $t_{s+1} \bmod q$ à partir de la valeur de $t_s \bmod q$, en utilisant directement l'équation (32.2).

RABIN-KARP consomme un temps de pré traitement $\Theta(m)$, et son temps de recherche de correspondances est $\Theta((n-m+1)m)$ dans le cas le plus défavorable, puisque (à l'instar de l'algorithme naïf de recherche de chaîne) l'algorithme de Rabin-Karp vérifie explicitement chaque décalage valide. Si $P = a^m$ et $T = a^n$, alors les vérifications prennent un temps $\Theta((n-m+1)m)$ vu que chacun des $n-m+1$ décalages possibles est valide.

Dans maintes applications, on s'attend à ce qu'il y ait un petit nombre de décalages valides (peut-être un certain nombre constant c) ; en pareil cas, le temps attendu de recherche de correspondances n'est que de $O((n-m+1) + cm) = O(n+m)$, plus le temps requis par le traitement des coups manqués. On peut baser une analyse heuristique sur l'hypothèse que la réduction des valeurs modulo q agit comme une transformation aléatoire entre Σ^* et \mathbf{Z}_q . (Voir l'étude de l'emploi de la division pour le hachage, à la section 11.3.1. Il est difficile de formaliser et prouver une telle hypothèse, bien qu'une approche viable consiste à supposer que q est choisi aléatoirement dans des entiers de taille idoine. Nous ne poursuivrons pas cette formalisation ici.) On peut alors s'attendre à ce que le nombre de coups manqués soit $O(n/q)$, car la chance qu'un t_s arbitraire soit équivalent à p , modulo q , peut être estimée à $1/q$. Comme il y a $O(n)$ positions pour lesquelles le test en ligne 10 échoue et que l'on consomme un temps $O(m)$ pour chaque coup, le temps espéré de recherche de correspondance de l'algorithme de Rabin-Karp est

$$O(n) + O(m(v + n/q)),$$

où v est le nombre de décalages valides. Ce temps d'exécution est $O(n)$ si $v = O(1)$ et si l'on prend $q \geq m$. En d'autres termes, si le nombre attendu de décalages valides est petit ($O(1)$) et que le nombre premier q est choisi de telle sorte qu'il soit plus grand que la longueur du motif recherché, alors on peut espérer que la procédure de Rabin-Karp ne consommera qu'un temps $O(n+m)$ pour la recherche de correspondance. Puisque $m \leq n$, ce temps attendu de recherche est $O(n)$.

Exercices

32.2.1 Si l'on travaille modulo $q = 11$, combien de fois l'algorithme de Rabin-Karp manquera-t-il son coup quand il recherchera la chaîne de caractères $P = 26$ dans le texte $T = 3141592653589793$?

32.2.2 Comment pourrait-on généraliser la méthode de Rabin-Karp pour rechercher dans une chaîne de texte une occurrence de n'importe laquelle des k chaînes de caractères d'un ensemble donné ? On commencera par supposer que les k motifs ont tous la même longueur. On généralisera ensuite la solution à des motifs de longueurs différentes.

32.2.3 Montrer comment généraliser la méthode de Rabin-Karp pour la recherche d'un motif $m \times m$ dans un tableau de caractères $n \times n$. (Le motif peut être décalé verticalement et horizontalement, mais il ne peut pas subir de rotation).

32.2.4 Alice possède un fichier (long) de n bits $A = \langle a_{n-1}, a_{n-2}, \dots, a_0 \rangle$ et Bob possède également un fichier de n bits $B = \langle b_{n-1}, b_{n-2}, \dots, b_0 \rangle$. Alice et Bob souhaitent savoir si leurs fichiers sont identiques. Pour éviter de transmettre A ou B entièrement, ils utilisent la vérification probabiliste ci-après. Ensemble, ils choisissent un nombre premier $q > 1000n$ et choisissent aléatoirement un entier x appartenant à $\{0, 1, \dots, q-1\}$. Puis, Alice évalue

$$A(x) = \left(\sum_{i=0}^n -1a_i x^i \right) \bmod q$$

et Bob évalue $B(x)$ de la même manière. Démontrer que, si $A \neq B$, il existe au plus une chance sur 1000 pour que $A(x) = B(x)$, alors que si les deux fichiers sont identiques, $A(x)$ est nécessairement égal à $B(x)$. (*Conseil* : Voir exercice 31.4.4.)

32.3 RECHERCHE DE CHAÎNE DE CARACTÈRES AU MOYEN D'AUTOMATES FINIS

De nombreux algorithmes de recherche de chaîne de caractères construisent un automate fini qui balaye la chaîne T à la recherche de toutes les occurrences de la chaîne P . Cette section présente une méthode de construction pour ce type d'automate. Ces automates de recherche de chaîne sont très efficaces : ils examinent chaque caractère du texte *une fois et une seule*, en prenant un temps constant pour chaque caractère. Le temps de recherche des correspondances (une fois l'automate construit) est donc $\Theta(n)$. Toutefois, le temps nécessaire à la construction de l'automate peut être grand si Σ est grand. La section 32.4 décrit un moyen astucieux de contourner le problème.

Nous commençons cette section par la définition d'un automate fini. Nous examinerons ensuite un automate spécial de recherche de chaîne et montrerons comment il peut servir à trouver des occurrences d'une chaîne dans un texte. Cette étude traitera de la façon de simuler le comportement d'un automate de recherche de chaîne sur un texte donné. Enfin, nous montrerons comment construire l'automate pour une chaîne de caractères donnée.

a) Automate fini

Un *automate fini* M est un quintuplet $(Q, q_0, A, \Sigma, \delta)$, où

– Q est un ensemble fini *d'états*,

- $q_0 \in Q$ est l'état initial,
- $A \subseteq Q$ est un ensemble distingué d'états terminaux,
- Σ est un alphabet fini,
- δ est une fonction de $Q \times \Sigma$ vers Q , appelée *fonction de transition* de M .

L'automate fini démarre à l'état q_0 et lit les caractères de la chaîne d'entrée un par un. Si l'automate se trouve dans l'état q et lit le caractère a , il passe (« effectue une transition ») de l'état q à l'état $\delta(q, a)$. Chaque fois que l'état courant q appartient à A , on dit que la machine M a *accepté* la chaîne lue jusqu'à cet endroit. On dit d'une entrée qui n'est pas acceptée qu'elle est *rejetée*. La figure 32.6 illustre ces définitions à l'aide d'un automate simple à deux états.

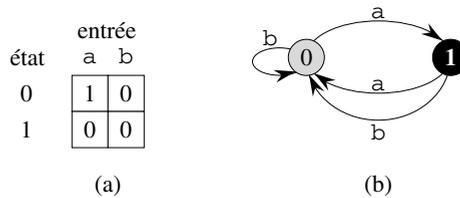


Figure 32.6 Un automate fini simple à deux états pour l'alphabet $\Sigma = \{a, b\}$, l'ensemble d'états $Q = \{0, 1\}$ et l'état initial $q_0 = 0$. (a) Une représentation tabulaire de la fonction de transition δ . (b) Un diagramme équivalent de transitions d'état. L'état 1 est le seul état d'acceptation (représenté en noir). Les arcs représentent les transitions. Par exemple, l'arc allant de l'état 1 à l'état 0 et étiqueté b indique $\delta(1, b) = 0$. Cet automate reconnaît les chaînes qui se terminent par un nombre impair de a . Plus précisément, une chaîne x est acceptée si et seulement si $x = yz$ avec $y = \varepsilon$ ou y se termine par b , et $z = a^k$ où k est impair. Par exemple, la séquence d'états dans lesquels entre cet automate pour l'entrée $abaaaa$ (état initial compris) est $\langle 0, 1, 0, 1, 0, 1 \rangle$. Cette entrée est donc acceptée. Pour l'entrée $abbaaa$, la séquence d'états est $\langle 0, 1, 0, 0, 1, 0 \rangle$ et elle est donc rejetée.

Un automate fini M induit une fonction ϕ , appelée *fonction d'état final*, de Σ^* vers Q telle que $\phi(w)$ est l'état dans lequel est M après avoir traité la chaîne w . Donc, M reconnaît une chaîne w si et seulement si $\phi(w) \in A$. La fonction ϕ est définie par la relation récursive

$$\begin{aligned} \phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{pour } w \in \Sigma^*, a \in \Sigma. \end{aligned}$$

b) Automates de recherche de motif

Il existe un automate de recherche pour chaque motif P ; cet automate doit être construit à partir du motif lors d'une étape de pré traitement, avant de pouvoir être utilisé pour chercher le motif dans une chaîne textuelle. La figure 32.7 illustre cette construction pour le motif $P = ababaca$. A partir de maintenant, nous supposons que P est une chaîne donnée fixée à l'avance ; pour alléger la notation, nous omettrons les références à P .

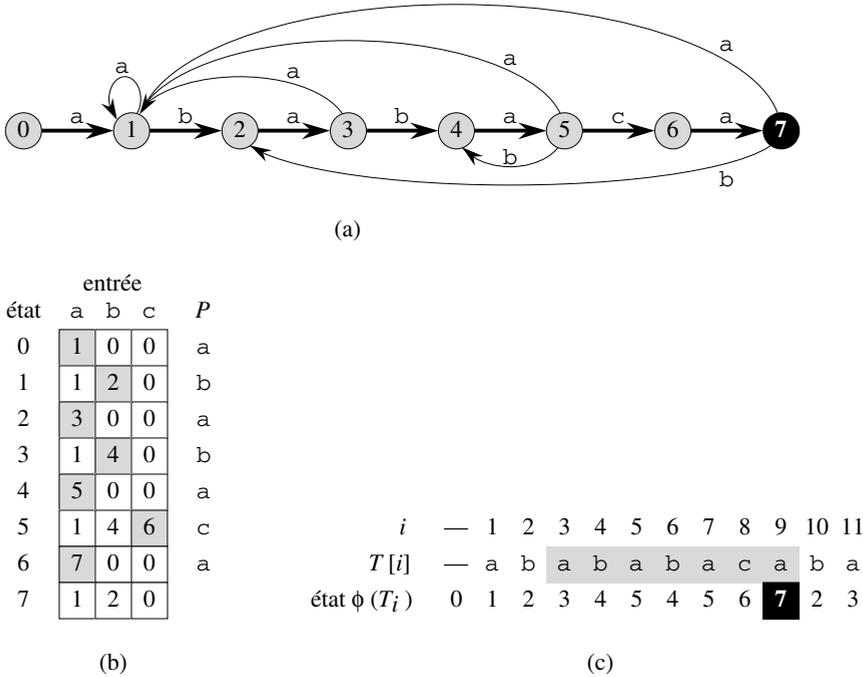


Figure 32.7 (a) Diagramme de transition d'état pour l'automate de recherche de chaîne qui accepte toutes les chaînes finissant par *ababaca*. L'état 0 est l'état initial et l'état 7 (en noir) est le seul état d'acceptation. Un arc étiqueté *a*, partant de l'état *i* et arrivant à l'état *j*, représente $\delta(i, a) = j$. Les arcs dirigés vers la droite forment le « squelette » de l'automate (dessiné en trait épais sur la figure) et correspondent aux comparaisons réussies entre le motif et les caractères d'entrée. Les arcs dirigés vers la gauche correspondent aux comparaisons ayant échoué. Certains de ces arcs ne sont pas représentés ; par convention, si un état *i* ne possède pas d'arc sortant étiqueté *a* pour un certain $a \in \Sigma$, alors $\delta(i, a) = 0$. (b) La fonction de transition δ correspondante et le motif $P = ababaca$. Les entrées correspondant à des comparaisons réussies entre le motif et les caractères d'entrée sont représentées en gris. (c) L'action de l'automate sur le texte $T = abababacaba$. Sous chaque caractère $T[i]$ du texte, on donne l'état $\phi(T_i)$ de l'automate après traitement du préfixe T_i . Une occurrence du motif est trouvée, qui se termine à la position 9.

Pour spécifier l'automate correspondant à une chaîne $P[1..m]$ donnée, on commence par définir une fonction auxiliaire σ , appelée *fonction suffixe* associée à P . La fonction σ est une application de Σ^* vers $\{0, 1, \dots, m\}$ telle que $\sigma(x)$ est la longueur du plus long préfixe de P qui est un suffixe de x :

$$\sigma(x) = \max \{k : P_k \sqsupseteq x\} .$$

La fonction suffixe σ est bien définie, car la chaîne vide $P_0 = \varepsilon$ est un suffixe de n'importe quelle chaîne. A titre d'exemple, pour la chaîne $P = ab$ on a $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$ et $\sigma(ccab) = 2$. Pour une chaîne P de longueur m , on a $\sigma(x) = m$ si et seulement si $P \sqsupseteq x$. On déduit de la définition de la fonction suffixe que, si $x \sqsupseteq y$, alors $\sigma(x) \leq \sigma(y)$.

On définit comme suit l'automate de recherche qui correspond à une chaîne $P[1..m]$ donnée.

- L'ensemble des états Q est $\{0, 1, \dots, m\}$. L'état initial q_0 est l'état 0 et l'état m est le seul état d'acceptation.
- La fonction de transition δ est définie par l'équation suivante, pour tout état q et tout caractère a :

$$\delta(q, a) = \sigma(P_q a) . \quad (32.3)$$

Voici un raisonnement intuitif pour la définition $\delta(q, a) = \sigma(P_q a)$. La machine conserve, au cours de son action, l'invariant

$$\phi(T_i) = \sigma(T_i) ; \quad (32.4)$$

Ce résultat est démontré par le théorème 32.4, donné plus loin. En clair, cela signifie qu'après l'analyse des i premiers caractères de la chaîne T , la machine se trouve dans l'état $\phi(T_i) = q$, où $q = \sigma(T_i)$ est la longueur du plus long suffixe de T_i qui est aussi un préfixe de la chaîne P . Si le prochain caractère analysé est $T[i+1] = a$, alors la machine devra effectuer une transition vers l'état $\sigma(T_{i+1}) = \sigma(T_i a)$. La démonstration du théorème montre que $\sigma(T_i a) = \sigma(P_q a)$. Autrement dit, pour calculer la longueur du plus long suffixe de $T_i a$ qui est un préfixe de P , on peut calculer le plus long suffixe de $P_q a$ qui est un préfixe de P . A chaque état, la machine n'a besoin de connaître que la longueur du plus long préfixe de P qui est aussi un suffixe de ce qui a été lu jusqu'ici. Donc, l'affectation $\delta(q, a) = \sigma(P_q a)$ conserve l'invariant (32.4) souhaité. Cet argument informel sera rendu plus rigoureux un peu plus tard.

Dans l'automate de recherche de chaîne de la figure 32.7, par exemple, on a $\delta(5, b) = 4$. On fait cette transition car, si l'automate lit un b dans l'état $q = 5$, alors $P_q b = ababab$ et le plus long préfixe de P qui est aussi un suffixe de $ababab$ est $P_4 = abab$.

Pour clarifier l'action d'un automate de recherche de chaîne, nous donnons à présent un programme simple et efficace qui simule le comportement de ce type d'automate (représenté par sa fonction de transition δ) lors de sa recherche des occurrences d'une chaîne P de longueur m dans un texte $T[1..n]$. Comme pour n'importe quel automate de recherche associé à une chaîne de longueur m , l'ensemble des états Q est $\{0, 1, \dots, m\}$, l'état initial est 0 et le seul état d'acceptation est l'état m .

RECHERCHE-AUTOMATE-FINI(T, δ, m)

```

1   $n \leftarrow \text{longueur}[T]$ 
2   $q \leftarrow 0$ 
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire  $q \leftarrow \delta(q, T[i])$ 
5          si  $q = m$ 
6              alors  $s \leftarrow i - m$ 
7              afficher « Le motif apparaît à la position »  $s$ 
```

La structure de boucle simple de RECHERCHE-AUTOMATE-FINI implique que son temps de recherche de correspondances sur un texte de longueur n est $\Theta(n)$. Ce temps ne prend toutefois pas en compte le temps de pré traitement requis pour calculer la fonction de transition δ . Ce problème est remis à plus tard, une fois que nous aurons prouvé le bon fonctionnement de la procédure RECHERCHE-AUTOMATE-FINI.

Considérons l'action de l'automate sur un texte d'entrée $T[1..n]$. Nous allons démontrer que l'automate se trouve dans l'état $\sigma(T_i)$ après avoir analysé le caractère $T[i]$. Puisque $\sigma(T_i) = m$ si et seulement si $P \sqsupseteq T_i$, la machine se trouve dans l'état d'acceptation m si et seulement si la chaîne P vient d'être analysée. Pour démontrer ce résultat, on utilise les deux lemmes qui suivent concernant la fonction suffixe σ .

Lemme 32.2 (Inégalité de la fonction suffixe) *Quels que soient la chaîne x et le caractère a , on a $\sigma(xa) \leq \sigma(x) + 1$.*

Démonstration : En se référant à la figure 32.8, soit $r = \sigma(xa)$. Si $r = 0$, la conclusion $r \leq \sigma(x) + 1$ est satisfaite de façon triviale d'après la non négativité de $\sigma(x)$. Supposons donc que $r > 0$. Or, $P_r \sqsupseteq xa$, d'après la définition de σ . Donc, $P_{r-1} \sqsupseteq x$, en supprimant le a situé à la fin de P_r et à la fin de xa . On a donc $r-1 \leq \sigma(x)$, puisque $\sigma(x)$ est le plus grand k tel que $P_k \sqsupseteq x$ et $\sigma(xa) = r \leq \sigma(x) + 1$. \square

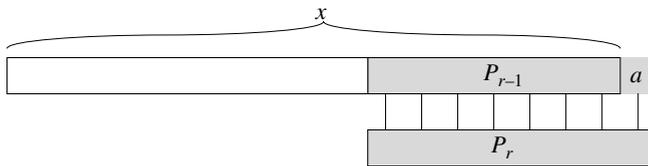


Figure 32.8 Une illustration de la démonstration du lemme 32.2. La figure montre que $r \leq \sigma(x) + 1$, où $r = \sigma(xa)$.

Lemme 32.3 (récursivité de la fonction suffixe) *Quels que soient la chaîne x et le caractère a , si $q = \sigma(x)$, alors $\sigma(xa) = \sigma(P_q a)$.*

Démonstration : D'après la définition de σ , on a $P_q \sqsupseteq x$. Comme le montre la figure 32.9, on a aussi $P_q a \sqsupseteq xa$. Si l'on prend $r = \sigma(xa)$, alors $r \leq q + 1$ d'après le lemme 32.2. Comme $P_q a \sqsupseteq xa$, $P_r \sqsupseteq xa$ et $|P_r| \leq |P_q a|$, le lemme 32.1 implique que $P_r \sqsupseteq P_q a$. Donc, $r \leq \sigma(P_q a)$; autrement dit, $\sigma(xa) \leq \sigma(P_q a)$. Mais on a également $\sigma(P_q a) \leq \sigma(xa)$, puisque $P_q a \sqsupseteq xa$. Donc $\sigma(xa) = \sigma(P_q a)$. \square

Nous sommes maintenant prêts pour démontrer notre théorème principal caractérisant le comportement d'un automate de recherche de chaîne sur un texte donné. Comme nous l'avons déjà dit, ce théorème montre que l'automate se contente de mémoriser, à chaque étape, le plus long préfixe du motif qui est un suffixe de ce qui a été lu jusqu'alors. Autrement dit, l'automate conserve l'invariant (32.4).

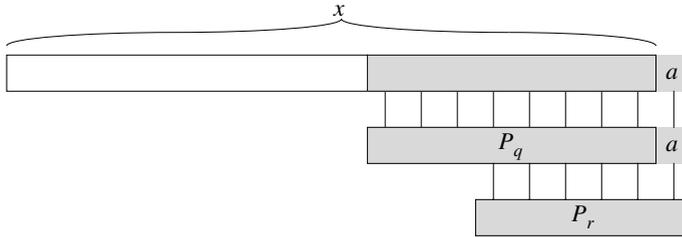


Figure 32.9 Une illustration de la démonstration du lemme 32.3. La figure montre que $r = \sigma(P_q a)$, où $q = \sigma(x)$ et $r = \sigma(xa)$.

Théorème 32.4 Si ϕ est la fonction d'état final d'un automate de recherche de chaîne associé à un motif P donné et si $T[1..n]$ est un texte d'entrée pour l'automate, alors

$$\text{pour } i = 0, 1, \dots, n. \quad \phi(T_i) = \sigma(T_i)$$

Démonstration : La démonstration se fait par récurrence sur i . Pour $i = 0$, le théorème est vrai puisque $T_0 = \varepsilon$. Donc, $\phi(T_0) = \sigma(T_0) = 0$.

On suppose à présent que $\phi(T_i) = \sigma(T_i)$ et on démontre que $\phi(T_{i+1}) = \sigma(T_{i+1})$. Posons $q = \phi(T_i)$ et $a = T[i+1]$. On a alors

$$\begin{aligned} \phi(T_{i+1}) &= \phi(T_i a) && \text{(d'après les définitions de } T_{i+1} \text{ et } a) \\ &= \delta(\phi(T_i), a) && \text{(d'après la définition de } \phi) \\ &= \delta(q, a) && \text{(d'après la définition de } q) \\ &= \sigma(P_q a) && \text{(d'après la définition (32.3) de } \delta) \\ &= \sigma(T_i a) && \text{(d'après le lemme 32.3 et la récurrence)} \\ &= \sigma(T_{i+1}) && \text{(d'après la définition de } T_{i+1}). \quad \square \end{aligned}$$

D'après le théorème 32.4, si la machine entre dans l'état q à la ligne 4, alors q est la plus grande valeur telle que $P_q \sqsupseteq T_i$. Donc, on a $q = m$ sur la ligne 5 si et seulement si une occurrence de la chaîne P vient d'être examinée. On en conclut que RECHERCHE-AUTOMATE-FINI se comporte de la manière attendue.

c) Calcul de la fonction de transition

La procédure ci-dessous calcule la fonction de transition δ à partir d'un motif donné $P[1..m]$.

CALCUL-FONCTION-TRANSITION(P, Σ)

```

1   $m \leftarrow \text{longueur}[P]$ 
2  pour  $q \leftarrow 0$  à  $m$ 
3      faire pour chaque caractère  $a \in \Sigma$ 
4          faire  $k \leftarrow \min(m + 1, q + 2)$ 
5              répéter  $k \leftarrow k - 1$ 
6                  jusqu'à  $P_k \sqsupseteq P_q a$ 
7                   $\delta(q, a) \leftarrow k$ 
8  retourner  $\delta$ 
```

Cette procédure calcule $\delta(q, a)$ directement, à partir de sa définition. Les boucles imbriquées des lignes 2 et 3 considèrent tous les états q et tous les caractères a , et les lignes 4–7 donnent à $\delta(q, a)$ la valeur du plus grand k tel que $P_k \sqsupseteq P_q a$. Le code commence avec la plus grande valeur possible pour k , soit $\min(m, q + 1)$, et fait diminuer k jusqu'à ce que $P_k \sqsupseteq P_q a$.

Le temps d'exécution de CALCUL-FONCTION-TRANSITION est $O(m^3 |\Sigma|)$; en effet, les boucles externes participent pour un facteur de $m |\Sigma|$, la boucle interne **répéter** peut s'exécuter au plus $m + 1$ fois et le test $P_k \sqsupseteq P_q a$ de la ligne 6 peut nécessiter de comparer jusqu'à m caractères. Des procédures beaucoup plus rapides existent; on peut améliorer le temps requis pour calculer δ à partir de P et le faire tomber à $O(m |\Sigma|)$ en utilisant certaines informations astucieusement calculées concernant la chaîne P (voir exercice 32.4.6). Avec cette procédure améliorée pour le calcul de δ , on peut trouver toutes les occurrences d'un motif de longueur m dans un texte de longueur n basé sur un alphabet Σ avec un temps de pré traitement $O(m |\Sigma|)$ et un temps de recherche de correspondances $\Theta(n)$.

Exercices

32.3.1 Construire l'automate de recherche du motif $P = \text{aabab}$ et illustrer son action sur le texte $T = \text{aaababaabaababaab}$.

32.3.2 Dessiner un diagramme de transition d'état pour un automate de recherche du motif $\text{ababbabbababbababbabb}$ sur l'alphabet $\Sigma = \{a, b\}$.

32.3.3 On dit d'une chaîne P qu'elle est *non recouvrable* si $P_k \sqsupseteq P_q$ implique $k = 0$ ou $k = q$. Décrire le diagramme de transition d'état de l'automate de recherche d'un motif non recouvrable.

32.3.4 * Étant données deux chaînes de caractères P et P' , décrire comment construire un automate fini qui détermine toutes les occurrences de l'une ou l'autre des deux chaînes. Essayer de minimiser le nombre d'états de votre automate.

32.3.5 Étant donnée une chaîne P contenant des jokers (voir exercice 32.1.4), montrer comment construire un automate fini qui soit capable de trouver une occurrence de P dans un texte T avec un temps de recherche de correspondances en $O(n)$, où $n = |T|$.

32.4 ALGORITHME DE KNUTH-MORRIS-PRATT

Nous présentons maintenant un algorithme, à temps linéaire, de recherche de chaîne dû à Knuth, Morris et Pratt. Cet algorithme permet d'éviter entièrement le calcul de la fonction de transition δ , et son temps de recherche de correspondances est $\Theta(n)$ grâce à une fonction auxiliaire $\pi[1..m]$ pré calculée à partir du motif en temps $\Theta(m)$. Le tableau π permet à la fonction de transition δ d'être calculée efficacement (au sens

amorti du terme) « à la volée » en fonction des besoins. Grosso modo, pour tout état $q = 0, 1, \dots, m$ et pour tout caractère $a \in \Sigma$, la valeur $\pi[q]$ contient les informations qui sont indépendantes de a et nécessaires pour calculer $\delta(q, a)$. (Cette remarque sera clarifiée bientôt.) Comme le tableau π n'a que m éléments alors que δ en a $\Theta(m|\Sigma|)$, on économise un facteur de $|\Sigma|$ dans le temps de pré traitement en calculant π à la place de δ .

a) Fonction préfixe d'une chaîne de caractères

La fonction préfixe d'une chaîne de caractères encapsule des informations sur la façon dont le motif concorde avec des décalages opérés sur lui-même. Ces données peuvent servir à éviter de tester des décalages inutiles dans l'algorithme de recherche naïf ou à éviter de précalculer δ pour un automate de recherche de chaîne.

Considérons l'action du comparateur naïf. La figure 32.10(a) montre un décalage s particulier d'un modèle contenant la chaîne $P = ababaca$, comparé à un texte T . Pour cet exemple, la correspondance a été établie pour $q = 5$ caractères, mais le 6ème caractère du motif n'est plus en concordance avec le caractère correspondant du texte. L'information selon laquelle q caractères ont concordé détermine les caractères correspondants dans le texte. Le fait de connaître ces q caractères du texte nous permet de déterminer immédiatement que certains décalages sont invalides. Sur la figure, le décalage $s + 1$ est obligatoirement invalide, puisque le premier caractère du motif (a), serait aligné avec un caractère du texte dont on sait qu'il correspond au deuxième caractère du motif (b). En revanche, le décalage $s + 2$ (montré sur la partie (b) de la figure) aligne les trois premiers caractères du motif avec trois caractères du texte qui correspondent nécessairement. D'une manière générale, il est utile de connaître la réponse à la question suivante :

Sachant que les caractères $P[1..q]$ du motif correspondent aux caractères $T[s+1..s+q]$ du texte, quel est le plus petit décalage $s' > s$ tel que

$$P[1..k] = T[s'+1..s'+k], \quad (32.5)$$

si $s' + k = s + q$?

Ce décalage s' est le premier décalage plus grand que s qui n'est pas invalide de façon certaine d'après notre connaissance de $T[s+1..s+q]$. Dans le meilleur cas, on a $s' = s + q$ et les décalages $s + 1, s + 2, \dots, s + q - 1$ peuvent tous être immédiatement éliminés. Dans tous les cas, au nouveau décalage s' on n'a pas besoin de comparer les k premiers caractères de P avec les caractères correspondants de T , puisqu'on est sûr qu'ils concordent d'après l'équation (32.5).

Les informations nécessaires peuvent être pré calculées en comparant le motif avec lui-même, comme illustré à la figure 32.10(c). Puisque $T[s'+1..s'+k]$ appartient à la partie connue du texte, c'est un suffixe de la chaîne P_q . L'équation (32.5) peut donc être interprétée comme la recherche du plus grand $k < q$ tel que $P_k \sqsupseteq P_q$. Donc, $s' = s + (q - k)$ est le prochain décalage potentiellement valide. Il est plus pratique,

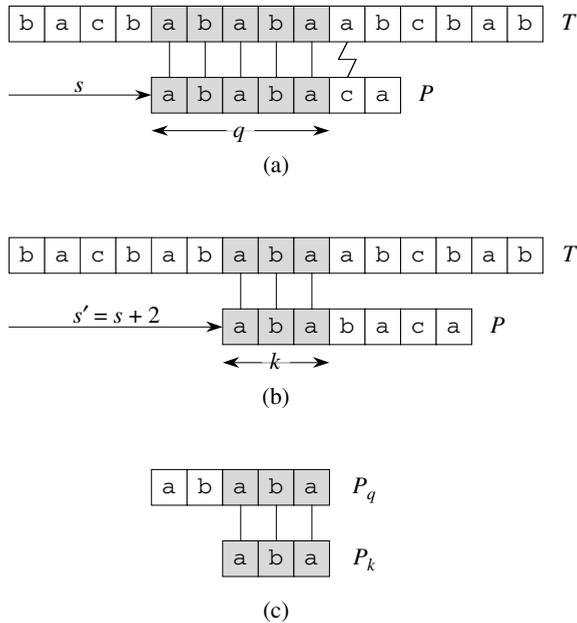


Figure 32.10 La fonction préfixe π . (a) La chaîne $P = ababaca$ est comparée à un texte T tel que les $q = 5$ premiers caractères concordent. Les caractères appariés, représentés en gris, sont reliés par des lignes verticales. (b) En ne se servant que de notre connaissance des 5 caractères concordants, on peut déduire qu'un décalage de $s+1$ est invalide, mais qu'un décalage de $s' = s+2$ est cohérent avec tout ce que l'on connaît du texte et donc est potentiellement valide. (c) Les informations utiles pour ces déductions peuvent être pré calculées en comparant le motif avec lui-même. Ici, on voit que le plus long préfixe de P qui est aussi un suffixe de P_5 est P_3 . Cette information est pré calculée et représentée dans le tableau π , de sorte que $\pi[5] = 3$. Sachant que q caractères ont été appariés avec succès au décalage s , le prochain décalage potentiellement valide est $s' = s + (q - \pi[q])$.

en fait, de stocker le nombre k de caractères qui concordent au nouveau décalage s' , plutôt que de stocker $s' - s$ par exemple. Cette information peut être utilisée pour accélérer à la fois l'algorithme naïf et l'automate fini.

Le pré calcul nécessaire peut être formalisé de la façon suivante. Étant donné un motif $P[1..m]$, la **fonction préfixe** de la chaîne P est la fonction $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m - 1\}$ telle que

$$\pi[q] = \max \{k : k < q \text{ et } P_k \sqsupseteq P_q\} .$$

Autrement dit, $\pi[q]$ est la longueur du plus long préfixe de P qui est un suffixe propre de P_q . À titre d'exemple supplémentaire, la figure 32.11 (a) donne la fonction préfixe π complète de la chaîne ababababca.

L'algorithme de recherche de Knuth-Morris-Pratt est donné en pseudo code sous la forme de la procédure KMP. Il s'inspire essentiellement de RECHERCHE-AUTOMATE-FINI, comme nous le verrons. KMP appelle la procédure auxiliaire CALCUL-FONCTION-PRÉFIXE pour calculer π .

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)

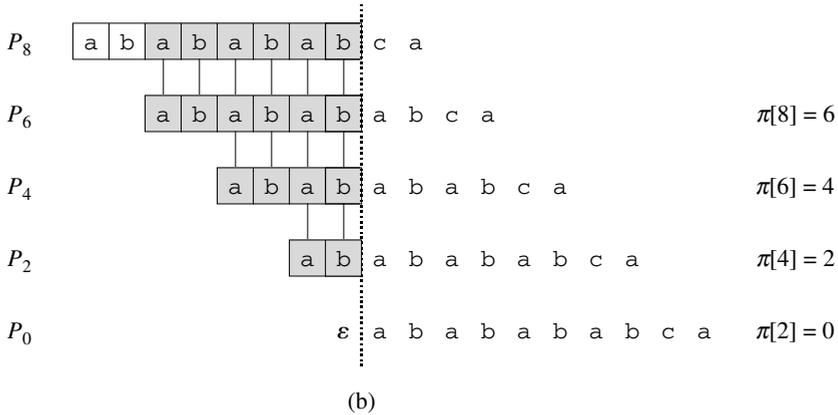


Figure 32.11 Une illustration du Lemme 32.5 pour la chaîne $P = ababababca$ et $q = 8$. (a) La fonction préfixe π pour la chaîne donnée. Puisque $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, et $\pi[2] = 0$, en itérant sur π nous obtenons $\pi^*[8] = \{6, 4, 2, 0\}$. (b) Nous décalons le modèle contenant la chaîne P vers la droite et notons le moment où un quelconque préfixe P_k de P vérifie un quelconque suffixe de P_8 ; ceci se produit pour $k = 6, 4, 2$, et 0 . Dans la figure, la première ligne donne P , et la ligne verticale en pointillé est tracée immédiatement après P_8 . Les lignes suivantes montre tout les décalages de P qui permette aux préfixes P_k de P de vérifier un suffixe P_8 . Les caractères satisfaisant le modèle sont montré en gris. Les lignes verticales relient les caractères trouvés alignés. Ainsi, $\{k : k < q \text{ et } P_k \sqsupseteq P_q\} = \{6, 4, 2, 0\}$. Le lemme affirme que, pour tout q , $\pi^*[q] = \{k : k < q \text{ et } P_k \sqsupseteq P_q\}$.

KMP(T, P)

```

1   $n \leftarrow \text{longueur}[T]$ 
2   $m \leftarrow \text{longueur}[P]$ 
3   $\pi \leftarrow \text{CALCUL-FONCTION-PRÉFIXE}(P)$ 
4   $q \leftarrow 0$  ▷ Nb de caractères concordant.
5  pour  $i \leftarrow 1$  à  $n$  ▷ Balaie le texte de la gauche vers la droite.
6    faire tant que  $q > 0$  et  $P[q+1] \neq T[i]$ 
7      faire  $q \leftarrow \pi[q]$  ▷ Prochain caractère ne concorde pas.
8    si  $P[q+1] = T[i]$ 
9      alors  $q \leftarrow q+1$  ▷ Prochain caractère concorde.
10   si  $q = m$  ▷ Est-ce que tout  $P$  a concordé ?
11     alors afficher « Le motif apparaît en position »  $i - m$ 
12      $q \leftarrow \pi[q]$  ▷ Chercher prochaine correspondance.

```

CALCUL-FONCTION-PRÉFIXE(P)

```

1   $m \leftarrow \text{longueur}[P]$ 
2   $\pi[1] \leftarrow 0$ 
3   $k \leftarrow 0$ 
4  pour  $q \leftarrow 2$  à  $m$ 
5      faire tant que  $k > 0$  et  $P[k+1] \neq P[q]$ 
6          faire  $k \leftarrow \pi[k]$ 
7          si  $P[k+1] = P[q]$ 
8              alors  $k \leftarrow k+1$ 
9           $\pi[q] \leftarrow k$ 
10 retourner  $\pi$ 

```

On commence par analyser le temps d'exécution de ces procédures. La démonstration de la validité de ces procédures sera plus compliquée.

b) Analyse du temps d'exécution

Le temps d'exécution de CALCUL-FONCTION-PRÉFIXE est $\Theta(m)$, si l'on utilise une analyse amortie basée sur la méthode du potentiel (voir section 17.3). On associe un potentiel k à l'état courant k de l'algorithme. Ce potentiel a une valeur initiale nulle, d'après la ligne 3. La ligne 6 fait diminuer k à chaque exécution, puisque $\pi[k] < k$. Cependant, comme $\pi[k] \geq 0$ pour tout k , k ne peut jamais devenir négatif. La seule autre ligne qui affecte k est la ligne 8, qui augmente k de un au plus à chaque exécution du corps de la boucle **pour**. Puisque $k < q$ à l'entrée de la boucle **pour** et que q est incrémenté dans chaque itération du corps de la boucle **pour**, l'inégalité $k < q$ est toujours vérifiée. (Cela justifie aussi l'affirmation que $\pi[q] < q$, d'après la ligne 9.) On peut payer chaque exécution du corps de la boucle **tant que** à la ligne 6 par la diminution correspondante de la fonction potentiel, car $\pi[k] < k$. La ligne 8 augmente la fonction potentiel d'un au plus, de sorte que le coût amorti du corps de la boucle aux lignes 5–9 est $O(1)$. Comme le nombre d'itérations de la boucle externe est $\Theta(m)$ et comme la valeur finale de la fonction potentiel est au moins aussi grande que sa valeur initiale, le temps d'exécution total réel de CALCUL-FONCTION-PRÉFIXE est $\Theta(m)$ dans le cas le plus défavorable.

Une analyse amortie similaire, utilisant la valeur de q comme fonction potentiel, montre que le temps de recherche des correspondances de KMP est $\Theta(n)$.

Comparé à RECHERCHE-AUTOMATE-FINI, l'utilisation de π à la place de δ permet de réduire le temps de pré traitement du motif, qui passe de $O(m|\Sigma|)$ à $\Theta(m)$, tout en conservant la borne $\Theta(n)$ pour le temps de recherche des correspondances.

c) Validité du calcul de la fonction préfixe

Nous commençons par un lemme essentiel qui montre que, en itérant la fonction préfixe π , on peut énumérer tous les préfixes P_k qui sont en même temps des suffixes

propres d'un préfixe P_q donné. Soit

$$\pi^*[q] = \{q, \pi[q], \pi^2[q], \pi^3[q], \dots, \pi^l[q]\},$$

où $\pi^i[q]$ est défini en termes d'itération fonctionnelle par $\pi^0[q] = q$ et $\pi^i[q] = \pi[\pi^{i-1}[q]]$ pour $i \geq 1$ sachant que la séquence dans $\pi^*[q]$ s'arrête quand $\pi^l[q] = 0$ est atteint. Le lemme suivant caractérise $\pi^*[q]$, comme illustré sur la figure 32.11.

Lemme 32.5 (Lemme de l'itération de la fonction préfixe) *Soit P une chaîne de longueur m et de fonction préfixe π . Alors, pour $q = 1, 2, \dots, m$, on a $\pi^*[q] = \{k : k < q \text{ et } P_k \sqsupset P_q\}$.*

Démonstration : Commençons par démontrer que

$$i \in \pi^*[q] \text{ implique } P_i \sqsupset P_q. \quad (32.6)$$

Si $i \in \pi^*[q]$, alors $i = \pi^u[q]$ pour un certain u positif. On démontre l'équation (32.6) par récurrence sur u . Pour $u = 1$, on a $i = \pi[q]$ et le lemme s'en déduit puisque $i < q$ et $P_{\pi[q]} \sqsupset P_q$.

Les relations $\pi[i] < i$ et $P_{\pi[i]} \sqsupset P_i$, la transitivité de $<$ et \sqsupset prouvent l'affirmation pour tout i in $\pi^*[q]$. Donc, $\pi^*[q] \subseteq \{k : k < q \text{ et } P_k \sqsupset P_q\}$.

Démontrons par l'absurde que $\{k : k < q \text{ et } P_k \sqsupset P_q\} \subseteq \pi^*[q]$. Supposons qu'il existe un entier dans l'ensemble $\{k : k < q \text{ et } P_k \sqsupset P_q\} - \pi^*[q]$, et soit j le plus grand de ces entiers. Comme $\pi[q]$ est la plus grande valeur de $\{k : k < q \text{ et } P_k \sqsupset P_q\}$ et que $\pi[q] \in \pi^*[q]$, on doit avoir $j < \pi[q]$; soit donc j' le plus petit entier de $\pi^*[q]$ qui est plus grand que j . (On peut prendre $j' = \pi[q]$ s'il n'y a pas d'autre nombre dans $\pi^*[q]$ qui est plus grand que j .) On a $P_j \sqsupset P_q$ car $j \in \{k : k < q \text{ et } P_k \sqsupset P_q\}$, et l'on a $P_{j'} \sqsupset P_q$ car $j' \in \pi^*[q]$. Donc, $P_j \sqsupset P_{j'}$ d'après le lemme 32.1, et j est la plus grande valeur inférieure à j' ayant cette propriété. Par conséquent, on a forcément $\pi[j'] = j$ et, comme $j' \in \pi^*[q]$, on a forcément $j \in \pi^*[q]$ aussi. Cette contradiction démontre le lemme. \square

L'algorithme CALCUL-FONCTION-PRÉFIXE calcule $\pi[q]$ dans l'ordre pour $q = 1, 2, \dots, m$. Le calcul de $\pi[1] = 0$ à la ligne 2 de CALCUL-FONCTION-PRÉFIXE est certainement correct, puisque $\pi[q] < q$ pour tout q . Le lemme suivant et son corollaire seront utilisés pour démontrer que CALCUL-FONCTION-PRÉFIXE calcule correctement $\pi[q]$ pour $q > 1$.

Lemme 32.6 *Soit P une chaîne de longueur m et soit π la fonction préfixe de P . Pour $q = 1, 2, \dots, m$, si $\pi[q] > 0$, alors $\pi[q] - 1 \in \pi^*[q - 1]$.*

Démonstration : Si $r = \pi[q] > 0$, alors $r < q$ et $P_r \sqsupset P_q$; donc, $r - 1 < q - 1$ et $P_{r-1} \sqsupset P_{q-1}$ (en supprimant le dernier caractère dans P_k et P_q). D'après le lemme 32.5, on a donc $\pi[q] - 1 = r - 1 \in \pi^*[q - 1]$. \square

Pour $q = 2, 3, \dots, m$, on définit le sous-ensemble $E_{q-1} \subseteq \pi^*[q-1]$ par

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ et } P_k \sqsupset P_{q-1} \text{ et } P[k+1] = P[q]\} \text{ (d'après le lemme 32.5)} \\ &= \{k : k < q-1 \text{ and } P_{k+1} \sqsupset P_q\} . \end{aligned}$$

L'ensemble E_{q-1} contient les valeurs $k < q-1$ pour lesquelles $P_k \sqsupset P_{q-1}$ et pour lesquelles $P_{k+1} \sqsupset P_q$, car $P[k+1] = P[q]$. Donc, E_{q-1} contient les valeurs de $k \in \pi^*[q-1]$ telles que l'on puisse étendre P_k à P_{k+1} et obtenir un suffixe propre de P_q .

Corollaire 32.7 Soit P une chaîne de longueur m et soit π la fonction préfixe de P . Pour $q = 2, 3, \dots, m$,

$$\pi[q] = \begin{cases} 0 & \text{si } E_{q-1} = \emptyset , \\ 1 + \max \{k \in E_{q-1}\} & \text{si } E_{q-1} \neq \emptyset . \end{cases}$$

Démonstration : Si E_{q-1} est vide, il n'y a pas de $k \in \pi^*[q-1]$ (y compris $k = 0$) pour lequel on peut étendre P_k à P_{k+1} et obtenir un suffixe propre de P_q . Donc, $\pi[q] = 0$.

Si E_{q-1} est non vide, alors pour chaque $k \in E_{q-1}$ on a $k+1 < q$ et $P_{k+1} \sqsupset P_q$. Donc, d'après la définition de $\pi[q]$, on a

$$\pi[q] \geq 1 + \max \{k \in E_{q-1}\} . \quad (32.7)$$

Notez que $\pi[q] > 0$. Soit $r = \pi[q] - 1$, de sorte que $r+1 = \pi[q]$. Comme $r+1 > 0$, on a $P[r+1] = P[q]$. En outre, d'après le lemme 32.6, on a $r \in \pi^*[q-1]$. Donc, $r \in E_{q-1}$ et donc $r \leq \max \{k \in E_{q-1}\}$ ou, de manière équivalente,

$$\pi[q] \leq 1 + \max \{k \in E_{q-1}\} . \quad (32.8)$$

La combinaison des équations (32.7) et (32.8) termine la démonstration. \square

Nous terminons enfin la démonstration de la validité du calcul de π par CALCUL-FONCTION-PRÉFIXE. Dans la procédure CALCUL-FONCTION-PRÉFIXE, au début de chaque itération de la boucle **pour** des lignes 4–9, on a $k = \pi[q-1]$. Cette condition est imposée par les lignes 2 et 3 à la première entrée dans la boucle, et elle reste vraie dans chaque itération successive grâce à la ligne 9. Les lignes 5–8 ajustent k pour qu'il devienne maintenant la valeur correcte de $\pi[q]$. La boucle des lignes 5–6 parcourt toutes les valeurs $k \in \pi^*[q-1]$ jusqu'à en trouver une pour laquelle $P[k+1] = P[q]$; à ce stade, k est la plus grande valeur de l'ensemble E_{q-1} et donc, d'après le corollaire 32.7, on peut donner à $\pi[q]$ la valeur $k+1$. Si l'on ne trouve pas un tel k , $k = 0$ en ligne 7. Si $P[1] = P[q]$, alors on doit donner à k et à $\pi[q]$ la valeur 1; sinon, on doit laisser k tel quel et donner à $\pi[q]$ la valeur 0. Les lignes 7–9 définissent correctement k et $\pi[q]$ dans l'un ou l'autre cas. Cela termine la démonstration de la conformité de CALCUL-FONCTION-PRÉFIXE.

d) Validité de l'algorithme KMP

On peut voir la procédure KMP comme une autre implémentation de la procédure RECHERCHE-AUTOMATE-FINI. En particulier, nous verrons que le code des lignes 6–9 de KMP est équivalent à la ligne 4 de RECHERCHE-AUTOMATE-FINI qui donne à q la valeur $\delta(q, T[i])$. Cependant, au lieu d'utiliser une valeur stockée de $\delta(q, T[i])$, cette valeur est recalculée, en fonction des besoins, à partir de π . Une fois que nous aurons établi que KMP simule le comportement de RECHERCHE-AUTOMATE-FINI, la validité de KMP se déduira de celle de RECHERCHE-AUTOMATE-FINI (nous verrons, toutefois, la nécessité de la ligne 12 de KMP).

La validité de KMP se déduit de l'affirmation selon laquelle soit $\delta(q, T[i]) = 0$, soit $\delta(q, T[i]) - 1 \in \pi^*[q]$. Pour vérifier cette affirmation, posons $k = \delta(q, T[i])$. Alors, $P_k \sqsubset P_q T[i]$ d'après les définitions de δ et σ . Donc, on a soit $k = 0$ soit $k \geq 1$ et $P_{k-1} \sqsubset P_q$ en supprimant le dernier caractère dans P_k et $P_q T[i]$ (auquel cas $k - 1 \in \pi^*[q]$). Par conséquent, on a soit $k = 0$ soit $k - 1 \in \pi^*[q]$, ce qui prouve l'assertion.

On se sert de cette affirmation de la façon suivante. Soit q' la valeur de q à l'entrée dans la ligne 6. On utilise l'équivalence $\pi^*[q] = \{k : k < q \text{ et } P_k \sqsubset P_q\}$ venant du lemme 32.5 pour justifier l'itération $q \leftarrow \pi[q]$ qui énumère les éléments de $\{k : P_k \sqsubset P_{q'}\}$. Les lignes 6–9 déterminent $\delta(q', T[i])$ en examinant les éléments de $\pi^*[q']$ dans l'ordre décroissant. Le code utilise l'assertion pour commencer avec $q = \phi(T_{i-1}) = \sigma(T_{i-1})$ et faire l'itération $q \leftarrow \pi[q]$ jusqu'à ce que soit trouvé un q tel que $q = 0$ ou que $P[q + 1] = T[i]$. Dans le premier cas, $\delta(q', T[i]) = 0$; dans le second cas, q est l'élément maximal de $E_{q'}$, de sorte que $\delta(q', T[i]) = q + 1$ d'après le corollaire 32.7.

La ligne 12 est nécessaire dans KMP pour éviter une possible référence à $P[m + 1]$ en ligne 6 après qu'une occurrence de P a été trouvée. (La preuve que $q = \sigma(T_{i-1})$ à l'exécution suivante de la ligne 6 reste vraie, d'après l'indication donnée à l'exercice 32.4.6 : $\delta(m, a) = \delta(\pi[m], a)$ ou, si l'on préfère, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ pour tout $a \in \Sigma$.) La preuve finale de la validité de l'algorithme de Knuth-Morris-Pratt vient de la validité de RECHERCHE-AUTOMATE-FINI, puisque nous voyons maintenant que KMP simule le comportement de RECHERCHE-AUTOMATE-FINI.

Exercices

32.4.1 Calculer la fonction préfixe π pour la chaîne de caractères `ababbabbababbababbabb` quand l'alphabet est $\Sigma = \{a, b\}$.

32.4.2 Donner un majorant de la taille de $\pi^*[q]$ sous la forme d'une fonction de q . Donner un exemple qui montre que c'est une borne serrée.

32.4.3 Expliquer comment déterminer les occurrences de la chaîne P dans le texte T , en examinant la fonction π de la chaîne PT (chaîne de longueur $m+n$, résultat de la concaténation de P et T).

32.4.4 Montrer comment améliorer KMP en remplaçant l'occurrence de π en ligne 7 (mais pas en ligne 12) par π' , où π' est défini récursivement pour $q = 1, 2, \dots, m$ par l'équation

$$\pi'[q] = \begin{cases} 0 & \text{si } \pi[q] = 0, \\ \pi'[\pi[q]] & \text{si } \pi[q] \neq 0 \text{ et } P[\pi[q] + 1] = P[q + 1], \\ \pi[q] & \text{si } \pi[q] \neq 0 \text{ et } P[\pi[q] + 1] \neq P[q + 1]. \end{cases}$$

Expliquer pourquoi l'algorithme modifié est correct et expliquer dans quel sens cette modification constitue une amélioration.

32.4.5 Donner un algorithme à temps linéaire qui détermine si un texte T est une rotation circulaire d'une autre chaîne T' . Par exemple, `arc` et `car` sont des rotations circulaires l'une de l'autre.

32.4.6 * Donner un algorithme efficace permettant de calculer la fonction de transition δ de l'automate de recherche correspondant à une chaîne P donnée. Votre algorithme devra s'exécuter en $O(m|\Sigma|)$. (*Conseil* : Montrer que $\delta(q, a) = \delta(\pi[q], a)$ si $q = m$ ou $P[q + 1] \neq a$.)

PROBLÈMES

32.1. Recherche de chaîne de caractères basée sur les facteurs de répétition

Soit y^i la concaténation de la chaîne y avec elle-même i fois. Par exemple, $(ab)^3 = ababab$. On dit qu'une chaîne $x \in \Sigma^*$ a un *facteur de répétition* de r si $x = y^r$ pour une certaine chaîne $y \in \Sigma^*$ et un certain $r > 0$. Soit $\rho(x)$ le plus grand r tel que x ait un facteur de répétition de r .

- Donner un algorithme efficace qui prend en paramètre une chaîne de caractères $P[1..m]$ et calcule $\rho(P_i)$ pour $i = 1, 2, \dots, m$. Quel est son temps d'exécution ?
- Pour une chaîne de caractères $P[1..m]$ quelconque, on définit $\rho^*(P)$ par $\max_{1 \leq i \leq m} \rho(P_i)$. Montrer que si la chaîne P est choisie aléatoirement dans l'ensemble de toutes les chaînes binaires de longueur m , alors la valeur attendue de $\rho^*(P)$ est $O(1)$.
- Montrer que l'algorithme de recherche de chaîne donné ci-après trouve bien toutes les occurrences de la chaîne P dans un texte $T[1..n]$ en un temps $O(\rho^*(P)n + m)$.

RECHERCHE-AVEC-RÉPÉTITION(P, T)

```

1   $m \leftarrow \text{longueur}[P]$ 
2   $n \leftarrow \text{longueur}[T]$ 
3   $k \leftarrow 1 + \rho^*(P)$ 
4   $q \leftarrow 0$ 
5   $s \leftarrow 0$ 
6  tant que  $s \leq n - m$ 
7      faire si  $T[s + q + 1] = P[q + 1]$ 
8          alors  $q \leftarrow q + 1$ 
9              si  $q = m$ 
10                 alors afficher « Le motif apparaî à la position »  $s$ 
11             si  $q = m$  ou  $T[s + q + 1] \neq P[q + 1]$ 
12                 alors  $s \leftarrow s + \max(1, \lceil q/k \rceil)$ 
13                  $q \leftarrow 0$ 

```

Cet algorithme est dû à Galil et Seiferas. En généralisant ces idées, ils obtiennent un algorithme de recherche de chaîne qui requiert un espace de stockage en $O(1)$ seulement, en plus de l'espace nécessaire à P et T .

NOTES

Les relations entre recherche de chaîne de caractères et théorie des automates finis sont traitées dans Aho, Hopcroft et Ullman [5]. L'algorithme de Knuth-Morris-Pratt [187] fut inventé indépendamment par Knuth et Pratt et par Morris ; leurs travaux furent publiés en commun. L'algorithme de Rabin-Karp fut proposé par Rabin et Karp [175]. Galil et Seiferas [107] donnent un algorithme intéressant de recherche de chaîne, déterministe et à temps linéaire, qui n'utilise qu'un espace $O(1)$ en plus de l'espace requis pour stocker la chaîne de caractères et le texte.

Chapitre 33

Géométrie algorithmique

La géométrie algorithmique est la branche de l'informatique qui étudie les algorithmes de résolution de problèmes géométriques. Dans l'ingénierie et les mathématiques modernes, la géométrie algorithmique trouve ses applications dans, entre autres domaines, l'infographie, la robotique, la conception des VLSI, la CAO et les statistiques. L'entrée d'un problème de géométrie algorithmique est, le plus souvent, la description d'un ensemble d'objets géométriques, tels un ensemble de points, un ensemble de segments de droite ou les sommets d'un polygone dans l'ordre trigonométrique. La sortie est souvent une réponse à une requête concernant les objets (par exemple, savoir si deux droites sont sécantes), parfois un nouvel objet géométrique, telle l'enveloppe convexe (plus petit polygone convexe circonscrit) d'un ensemble de points.

Dans ce chapitre, nous étudierons quelques algorithmes de géométrie algorithmique en dimension deux, c'est-à-dire dans le plan. Chaque objet d'entrée est représenté par un ensemble de points $\{p_i\}$, où chaque $p_i = (x_i, y_i)$ et $x_i, y_i \in \mathbf{R}$. Par exemple, un polygone P à n sommets est représenté par la suite $\langle p_0, p_1, p_2, \dots, p_{n-1} \rangle$ de ses sommets, par ordre de leur apparition sur le contour de P . La géométrie algorithmique peut aussi s'intéresser à des espaces de dimension trois ou plus, mais ces problèmes et leurs solutions peuvent être très difficiles à visualiser. En se restreignant à deux dimensions, on peut déjà avoir un bon échantillon des techniques de la géométrie algorithmique.

La section 33.1 explique comment répondre avec efficacité et précision à des questions élémentaires concernant les segments de droite : faut-il tourner dans le sens des aiguilles d'une montre ou dans le sens trigonométrique pour rencontrer un segment

en partant d'un autre, quand ils ont une extrémité commune ? ; dans quelle direction tourne-t-on quand on traverse deux segments contigus ; deux segments de droite sont-ils sécants ? La section 33.2 présente une technique appelée « balayage », qui nous servira à développer un algorithme en temps $O(n \lg n)$ qui permet de déterminer s'il existe des intersections parmi un ensemble de n segments de droite. La section 33.3 donne deux algorithmes de « balayage » circulaire « qui calculent l'enveloppe convexe (plus petit polygone convexe circonscrit) d'un ensemble de n points : le balayage de Graham qui s'exécute en $O(n \lg n)$ et le parcours de Jarvis qui s'exécute en temps $O(nh)$, où h est le nombre de sommets de l'enveloppe convexe. Enfin, la section 33.4 donne un algorithme diviser-pour-régner en temps $O(n \lg n)$ qui trouve la paire de points les plus proches dans un ensemble de n points du plan.

33.1 PROPRIÉTÉS DES SEGMENTS DE DROITE

Dans ce chapitre, il y aura plusieurs algorithmes qui exigeront des réponses à des questions sur les propriétés des segments de droite. Une **combinaison linéaire convexe** de deux points distincts $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$ est un point $p_3 = (x_3, y_3)$ tel que, pour un certain α de l'intervalle $0 \leq \alpha \leq 1$, on a $x_3 = \alpha x_1 + (1 - \alpha)x_2$ et $y_3 = \alpha y_1 + (1 - \alpha)y_2$. On écrit aussi que $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitivement, p_3 est un point de la droite passant par p_1 et p_2 et se trouve sur ou entre les points p_1 et p_2 sur la droite. Étant donnés deux points distincts p_1 et p_2 , le **segment de droite** $\overline{p_1 p_2}$ est l'ensemble des combinaisons linéaires convexes de p_1 et p_2 . Les points p_1 et p_2 sont appelés **extrémités** du segment $\overline{p_1 p_2}$. Parfois, l'ordre de p_1 et p_2 est important et on parle alors de **segment orienté** $\overrightarrow{p_1 p_2}$. Si p_1 est l'**origine** $(0, 0)$, on peut considérer le segment orienté $\overrightarrow{p_1 p_2}$ comme le **vecteur** p_2 .

Dans cette section, nous nous intéresserons aux problèmes suivants :

- 1) Étant donnés deux segments orientés $\overrightarrow{p_0 p_1}$ et $\overrightarrow{p_0 p_2}$, le segment $\overrightarrow{p_0 p_1}$ est-il situé dans le sens des aiguilles d'une montre par rapport à $\overrightarrow{p_0 p_2}$?
- 2) Étant donnés deux segments de droite $\overline{p_0 p_1}$ et $\overline{p_1 p_2}$, si l'on traverse $\overline{p_0 p_1}$ puis $\overline{p_1 p_2}$, la bifurcation au point p_1 se fait-elle vers la gauche ?
- 3) Les segments de droite $\overline{p_1 p_2}$ et $\overline{p_3 p_4}$ sont-ils sécants ?

Il n'existe aucune restriction sur les points donnés.

Il est possible de répondre à chaque problème en temps $O(1)$, ce qui n'est pas surprenant puisque la taille de l'entrée de chaque question est $O(1)$. Par ailleurs, nos méthodes ne feront appel qu'à des additions, soustractions, multiplications et comparaisons. Nous n'avons besoin ni de divisions ni de fonctions trigonométriques, qui peuvent être coûteuses en calcul et conduire à des erreurs d'arrondi. Par exemple, la méthode « directe » pour déterminer si deux segments sont sécants (on calcule l'équation de la droite $y = mx + b$ pour chaque segment (m est la pente et b l'ordonnée

à l'origine), on recherche le point d'intersection des droites et l'on vérifie que le point appartient aux deux segments) utilise la division pour trouver le point d'intersection. Lorsque les segments sont presque parallèles, cette méthode est très sensible à la précision de l'opération division sur les ordinateurs réels. La méthode de cette section, qui évite la division, est beaucoup plus précise.

a) Produits en croix

Le calcul de produits en croix est la clé de voûte de nos méthodes pour segments de droite. On considère les vecteurs p_1 et p_2 , montrés à la figure 33.1(a). Le **produit en croix** $p_1 \times p_2$ peut être interprété comme étant l'aire signée du parallélogramme formé par les points $(0, 0)$, p_1 , p_2 et $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. Une définition équivalente, mais plus utile, fait du produit en croix le déterminant d'une matrice ⁽¹⁾ :

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 . \end{aligned}$$

Si $p_1 \times p_2$ est positif, alors p_1 est placé dans le sens des aiguilles d'une montre par rapport à p_2 en supposant que l'on tourne autour de l'origine $(0, 0)$; si le produit en croix est négatif, alors p_1 est placé dans le sens contraire des aiguilles d'une montre par rapport à p_2 . La figure 33.1(b) montre les régions sens-des-aiguilles-d'une-montre et sens-contre-aiguilles-relativement à un vecteur p . Une condition limite survient quand le produit en croix vaut zéro ; dans ce cas, les vecteurs sont **colinéaires** et pointent soit dans le même sens, soit dans des sens opposés.

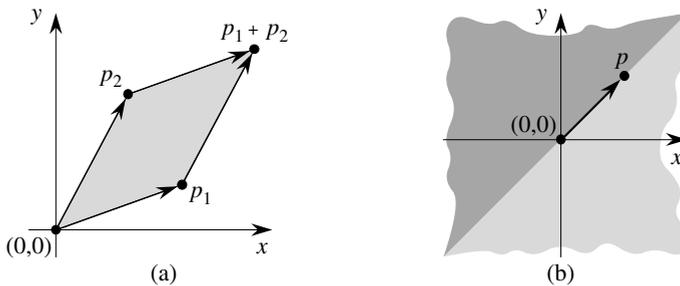


Figure 33.1 (a) Le produit en croix des vecteurs p_1 et p_2 est l'aire signée du parallélogramme. (b) La région claire contient les vecteurs qui se trouvent dans le sens des aiguilles d'une montre par rapport à p . La région foncée contient les vecteurs qui se trouvent dans le sens contraire des aiguilles d'une montre par rapport à p .

(1) En réalité, le produit en croix est un concept tridimensionnel. C'est un vecteur perpendiculaire à la fois à p_1 et p_2 , au sens de la « règle » des trois « doigts » et dont le module est égal à $|x_1 y_2 - x_2 y_1|$. Toutefois, on verra qu'il est plus commode dans ce chapitre de considérer le produit en croix comme étant la valeur $x_1 y_2 - x_2 y_1$.

Pour déterminer si la rotation vers un segment orienté $\overrightarrow{p_0p_1}$ à partir d'un segment orienté $\overrightarrow{p_0p_2}$ autour de leur extrémité commune p_0 se fait dans le sens des aiguilles d'une montre, on fait simplement une translation de l'origine au point p_0 . Autrement dit, on appelle $p'_1 = (x'_1, y'_1)$ le vecteur $p_1 - p_0$, où $x'_1 = x_1 - x_0$ et $y'_1 = y_1 - y_0$ et on définit $p_2 - p_0$ de la même façon. On calcule ensuite le produit en croix

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) .$$

Si ce produit en croix est positif, alors $\overrightarrow{p_0p_1}$ est atteint par une rotation dans le sens des aiguilles d'une montre à partir de $\overrightarrow{p_0p_2}$; s'il est négatif, la rotation s'effectue dans le sens inverse des aiguilles d'une montre.

b) Déterminer si des segments consécutifs tournent à gauche ou à droite

La question suivante sera de savoir si deux segments de droite consécutifs $\overrightarrow{p_0p_1}$ et $\overrightarrow{p_1p_2}$ tournent à droite ou à gauche au point p_1 . Autrement dit, on voudrait une méthode pour déterminer dans quel sens est orienté un angle donné $\angle p_0p_1p_2$. Les produits en croix nous permettent de répondre à cette question sans calculer l'angle. Comme on peut le voir sur la figure 33.2, on se contente de vérifier si le segment orienté $\overrightarrow{p_0p_2}$ se situe dans le sens des aiguilles d'une montre (sens trigonométrique) par rapport au segment orienté $\overrightarrow{p_0p_1}$. Pour cela, on calcule le produit en croix $(p_2 - p_0) \times (p_1 - p_0)$. Si le signe du produit en croix est négatif, alors $\overrightarrow{p_0p_2}$ est dans le sens contraire des aiguilles d'une montre (sens trigonométrique) par rapport à $\overrightarrow{p_0p_1}$ et donc on tourne donc à gauche au point p_1 . Un produit en croix positif indique une orientation dans le sens des aiguilles d'une montre et une bifurcation vers la droite. Un produit en croix égal à 0 signifie que les points p_0, p_1 et p_2 sont colinéaires.

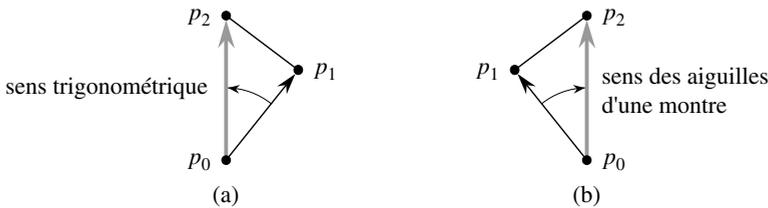


Figure 33.2 Grâce au produit en croix, on détermine dans quel sens tournent les segments de droite consécutifs $\overrightarrow{p_0p_1}$ et $\overrightarrow{p_1p_2}$ au point p_1 . On vérifie si le segment orienté $\overrightarrow{p_0p_2}$ est dans le sens des aiguilles d'une montre ou dans le sens contraire par rapport au segment orienté $\overrightarrow{p_0p_1}$. (a) Si c'est le sens contraire des aiguilles d'une montre, la bifurcation s'effectue vers la gauche. (b) Si c'est le sens des aiguilles d'une montre, elle s'effectue vers la droite.

c) Déterminer si deux segments de droite sont sécants

Pour savoir si deux segments se coupent, on voit si chacun traverse la droite support de l'autre. Un segment $\overrightarrow{p_1p_2}$ *traverse* une droite si le point p_1 est d'un côté de la droite et le point p_2 de l'autre côté. Il y a apparition d'un cas limite quand p_1 ou p_2 est sur la droite elle-même. Deux segments se coupent si et seulement si l'une au moins des deux conditions suivantes est vérifiée :

- 1) Chaque segment traverse la droite contenant l'autre.
- 2) Une extrémité d'un segment appartient à l'autre segment. (Cette condition vient du cas limite.)

Les procédures suivantes implémentent ce concept. INTERSECTION-SEGMENTS retourne VRAI si les segments $\overline{p_1p_2}$ et $\overline{p_3p_4}$ se coupent, et FAUX s'ils ne se coupent pas. Elle appelle les sous-routines DIRECTION, qui calcule les orientations relatives à l'aide de la méthode du produit en croix précédemment exposée, et SUR-SEGMENT, qui vérifie si un point, dont on sait qu'il est colinéaire à un segment, appartient à ce segment.

INTERSECTION-SEGMENTS(p_1, p_2, p_3, p_4)

```

1   $d_1 \leftarrow \text{DIRECTION}(p_3, p_4, p_1)$ 
2   $d_2 \leftarrow \text{DIRECTION}(p_3, p_4, p_2)$ 
3   $d_3 \leftarrow \text{DIRECTION}(p_1, p_2, p_3)$ 
4   $d_4 \leftarrow \text{DIRECTION}(p_1, p_2, p_4)$ 
5  si (( $d_1 > 0$  et  $d_2 < 0$ ) ou ( $d_1 < 0$  et  $d_2 > 0$ )) et
      (( $d_3 > 0$  et  $d_4 < 0$ ) ou ( $d_3 < 0$  et  $d_4 > 0$ ))
6    alors retourner VRAI
7  sinon si  $d_1 = 0$  et SUR-SEGMENT( $p_3, p_4, p_1$ )
8    alors retourner VRAI
9  sinon si  $d_2 = 0$  et SUR-SEGMENT( $p_3, p_4, p_2$ )
10   alors retourner VRAI
11 sinon si  $d_3 = 0$  et SUR-SEGMENT( $p_1, p_2, p_3$ )
12   alors retourner VRAI
13 sinon si  $d_4 = 0$  et SUR-SEGMENT( $p_1, p_2, p_4$ )
14   alors retourner VRAI
15 sinon retourner FAUX

```

DIRECTION(p_i, p_j, p_k)

```

1  retourner  $(p_k - p_i) \times (p_j - p_i)$ 

```

SUR-SEGMENT(p_i, p_j, p_k)

```

1  si  $\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j)$  et  $\min(y_i, y_j) \leq y_k \leq \max(y_i, y_j)$ 
2    alors retourner VRAI
3  sinon retourner FAUX

```

INTERSECTION-SEGMENTS fonctionne comme suit. Les lignes 1–4 calculent l'orientation relative d_i de chaque extrémité p_i par rapport à l'autre segment. Si toutes les orientations relatives sont non nulles, alors il est facile de déterminer si les segments $\overline{p_1p_2}$ et $\overline{p_3p_4}$ se coupent. Voici la méthode : le segment $\overline{p_1p_2}$ traverse la droite support du segment $\overline{p_3p_4}$ si les segments orientés $\overrightarrow{p_3p_1}$ et $\overrightarrow{p_3p_2}$ ont des orientations opposées relativement à $\overrightarrow{p_3p_4}$. En pareil cas, les signes de d_1 et d_2 diffèrent. De même, $\overline{p_3p_4}$ traverse la droite support de $\overline{p_1p_2}$ si les signes de d_3 et d_4 diffèrent.

Si le test en ligne 5 est vrai, alors les segments se coupent et INTERSECTION-SEGMENTS retourne VRAI. La figure 33.3(a) illustre ce cas. Sinon, les segments ne traversent pas la droite support de l'autre, encore qu'il puisse y avoir un cas limite. Si toutes les orientations relatives sont non nulles, il n'y a pas de cas limite. Tous les tests par rapport à 0 faits aux lignes 7–13 échouent alors, et INTERSECTION-SEGMENTS retourne FAUX en ligne 15. La figure 33.3(b) illustre ce cas.

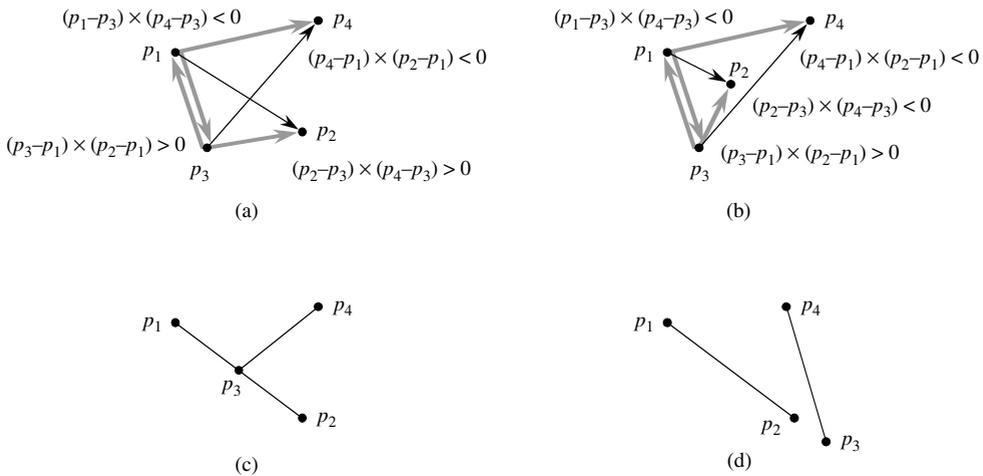


Figure 33.3 Cas de la procédure INTERSECTION-SEGMENTS. (a) Les segments $\overline{p_1p_2}$ et $\overline{p_3p_4}$ traversent chacun la droite support de l'autre. Comme $\overline{p_3p_4}$ traverse la droite support de $\overline{p_1p_2}$, les signes des produits en croix $(p_3 - p_1) \times (p_2 - p_1)$ et $(p_4 - p_1) \times (p_2 - p_1)$ diffèrent. Comme $\overline{p_1p_2}$ traverse la droite support de $\overline{p_3p_4}$, les signes des produits en croix $(p_1 - p_3) \times (p_4 - p_3)$ et $(p_2 - p_3) \times (p_4 - p_3)$ diffèrent. (b) Le segment $\overline{p_3p_4}$ traverse la droite support de $\overline{p_1p_2}$, mais $\overline{p_1p_2}$ ne traverse pas la droite support de $\overline{p_3p_4}$. Les signes des produits en croix $(p_1 - p_3) \times (p_4 - p_3)$ et $(p_2 - p_3) \times (p_4 - p_3)$ sont les mêmes. (c) Le point p_3 est colinéaire à $\overline{p_1p_2}$ et se trouve entre p_1 et p_2 . (d) Le point p_3 est colinéaire à $\overline{p_1p_2}$, mais ne se trouve pas entre p_1 and p_2 . Les segments ne se coupent pas.

Il y a un cas limite si toutes les orientations relatives d_k valent 0. Ici, on sait que p_k est colinéaire avec l'autre segment. Il est directement sur l'autre segment si et seulement si il est entre les extrémités de l'autre segment. La procédure SUR-SEGMENT retourne une valeur indiquant si p_k est entre les extrémités du segment $\overline{p_i p_j}$, qui sera l'autre segment dans l'appel des lignes 7–13 ; la procédure part du principe que p_k est colinéaire au segment $\overline{p_i p_j}$. Les figures 33.3(c) et (d) montrent des cas de points colinéaires. Sur la figure 33.3(c), p_3 est sur $\overline{p_1 p_2}$, et donc INTERSECTION-SEGMENTS retourne VRAI en ligne 12. Aucune extrémité n'est sur d'autres segments à la figure 33.3(d), et donc INTERSECTION-SEGMENTS retourne FAUX en ligne 15.

d) Autres applications des produits en croix

Les sections suivantes présenteront d'autres applications des produits en croix. À la section 33.3, on aura besoin de trier un ensemble de points en fonction de leurs angles polaires par rapport à une certaine origine. Ainsi que l'exercice 33.1.3 vous demandera de le montrer, les produits en croix peuvent servir à effectuer les comparaisons dans la procédure de tri. À la section 33.2, on utilisera des arbres rouge-noir pour gérer l'ordre vertical d'un ensemble de segments de droite. Au lieu de gérer des valeurs de clé explicites, on remplacera chaque comparaison de clé dans le code de l'arbre rouge-noir par un calcul de produit en croix pour déterminer lequel de deux segments qui coupent une droite verticale donnée est au-dessus de l'autre.

Exercices

33.1.1 Démontrer que, si $p_1 \times p_2$ est positif, alors le vecteur p_1 est orienté dans le sens des aiguilles d'une montre par rapport au vecteur p_2 et par rapport à l'origine $(0, 0)$; montrer que, si le produit en croix est négatif, alors p_1 est orienté dans le sens contraire des aiguilles d'une montre par rapport à p_2 .

33.1.2 Le professeur Trissotin affirme qu'il suffit de tester la dimension x sur la ligne 1 de SUR-SEGMENT. Montrer pourquoi le professeur mérite son nom.

33.1.3 L'*angle polaire* d'un point p_1 par rapport à une origine p_0 est l'angle du vecteur $p_1 - p_0$ dans le système usuel de coordonnées polaires. Ainsi, l'angle polaire de $(3, 5)$ par rapport à $(2, 4)$ est l'angle du vecteur $(1, 1)$, soit 45 degrés, soit encore $\pi/4$ radians. L'angle polaire de $(3, 3)$ par rapport à $(2, 4)$ est l'angle du vecteur $(1, -1)$, soit 315 degrés, soit encore $7\pi/4$ radians. Écrire un pseudo code qui trie une séquence $\langle p_1, p_2, \dots, p_n \rangle$ de n points selon leurs angles polaires par rapport à un point origine donné p_0 . Votre procédure devra prendre un temps $O(n \lg n)$ et faire appel aux produits en croix pour comparer les angles.

33.1.4 Montrer comment déterminer en temps $O(n^2 \lg n)$ si trois points quelconques d'un ensemble de n points sont colinéaires.

33.1.5 Un *polygone* est une courbe plane, refermée sur elle-même et composée d'une suite de segments de droite appelés *côtés* du polygone. Un point reliant deux côtés consécutifs est dit *sommet* du polygone. Si le polygone est *simple*, ce qui sera généralement le cas, il ne se recoupe pas lui-même. L'ensemble des points du plan englobés par un polygone simple forme l'*intérieur* du polygone, l'ensemble des points du polygone lui-même forme le *contour* et l'ensemble des points entourant le polygone forme son *extérieur*. Un polygone simple est *convexe* si, étant donnés deux points quelconques situés sur le contour ou à l'intérieur, tous les points du segment de droite reliant ces deux points se trouvent sur le contour ou à l'intérieur du polygone. Le professeur Amundsen propose la méthode suivante pour déterminer si une séquence $\langle p_0, p_1, \dots, p_{n-1} \rangle$ de n points forme l'ensemble des sommets consécutifs d'un polygone convexe. On sort « oui » si l'ensemble $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \dots, n-1\}$, où l'addition indiciée est effectuée modulo n , ne contient pas en même temps des bifurcations

vers la droite et vers la gauche ; sinon, on sort « non ». Montrer que cette méthode s'exécute en temps linéaire, mais qu'elle ne produit pas toujours la bonne réponse. Modifier la méthode du professeur pour qu'elle produise toujours la réponse correcte et en temps linéaire.

33.1.6 Étant donné un point $p_0 = (x_0, y_0)$, le **rayon horizontal droit** à partir de p_0 est l'ensemble des points $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ et } y_i = y_0\}$, c'est-à-dire l'ensemble des points situés à droite de p_0 , y compris p_0 lui-même. Montrer comment déterminer en temps $O(1)$ si un rayon horizontal droit partant de p_0 coupe un segment de droite $\overline{p_1p_2}$, en ramenant le problème à celui de savoir si deux segments de droite sont sécants.

33.1.7 Une façon de déterminer si un point p_0 se trouve à l'intérieur d'un polygone P simple pas forcément convexe, consiste à regarder un rayon quelconque partant de p_0 et à vérifier que le rayon coupe le contour de P un nombre impair de fois mais que p_0 lui-même ne se trouve pas sur le contour de P . Montrer comment déterminer en temps $\Theta(n)$ si un point p_0 se trouve à l'intérieur d'un polygone P à n sommets. (*conseil* : Utiliser l'exercice 33.1.6. Vérifier la validité de l'algorithme quand le rayon coupe le contour du polygone en un sommet et quand le rayon recouvre un côté du polygone.)

33.1.8 Montrer comment calculer en temps $\Theta(n)$ la surface d'un polygone simple, pas forcément convexe, à n sommets. (Voir exercice 33.1.5 pour les définitions relatives aux polygones.)

33.2 DÉTERMINER SI DEUX SEGMENTS DONNÉS SE COUPENT

Cette section présente un algorithme qui détermine si deux segments quelconques d'un ensemble de segments de droite sont sécants. L'algorithme utilise une technique appelée « balayage », commune à beaucoup d'algorithmes de géométrie algorithmique. Par ailleurs, comme le montrent les exercices de la fin de cette section, cet algorithme, ou ses variantes simples, peuvent servir à résoudre d'autres problèmes de géométrie algorithmique.

L'algorithme s'exécute en temps $O(n \lg n)$, où n est le nombre de segments que l'on se donne. Il se contente de dire si une intersection existe ; il n'affiche pas toutes les intersections. (D'après l'exercice 33.2.1, la découverte de *toutes* les intersections dans un ensemble de n segments demande un temps $\Omega(n^2)$ dans le cas le plus défavorable.)

Dans le *balayage*, une *droite de balayage* verticale imaginaire se déplace à travers l'ensemble donné d'objets géométriques, en général de la gauche vers la droite. La dimension sur laquelle se déplace la droite de balayage, ici la dimension x , est considérée comme une dimension de temps. Le balayage permet de trier les objets géométriques, le plus souvent en les plaçant dans une structure de données dynamique, et d'exploiter les relations qu'ils ont entre eux. L'algorithme d'intersection de

segments donné dans cette section examine les extrémités des segments de la gauche vers la droite et vérifie l'existence d'une intersection chaque fois qu'il rencontre une extrémité.

Pour décrire et valider notre algorithme de recherche d'intersection de deux segments parmi n , nous ferons deux hypothèses simplificatrices. Primo, on suppose qu'aucun segment n'est vertical. Secundo, on suppose que trois segments ne peuvent pas se couper en un même point. (Les exercices 33.2.8 et 33.2.9 vous demanderont de montrer que l'algorithme est suffisamment robuste pour qu'une petite modification du code permette de se passer de ces simplifications.) En fait, supprimer ce type de simplifications et gérer les conditions aux limites représente souvent la partie la plus difficile de la programmation des algorithmes de géométrie algorithmique et de leur validation.

a) Tri de segments

Puisqu'on suppose qu'il n'existe aucun segment vertical, tout segment qui croise une droite de balayage verticale ne le croise qu'en un seul point. On peut donc ranger les segments qui coupent une droite de balayage verticale en fonction des ordonnées des points d'intersection.

Plus précisément, soient deux segments s_1 et s_2 . On dit que ces segments sont **comparables** en x si la droite de balayage verticale d'abscisse x les coupe tous les deux. On dit que s_1 est **au-dessus** de s_2 en x , ce qui s'écrit $s_1 >_x s_2$, si s_1 et s_2 sont comparables en x et si l'intersection de s_1 et de la droite de balayage en x est supérieure à l'intersection de s_2 et de la même droite. A la figure 33.4(a), par exemple, on a les relations $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ et $b >_u c$. Le segment d n'est comparable avec aucun autre segment.

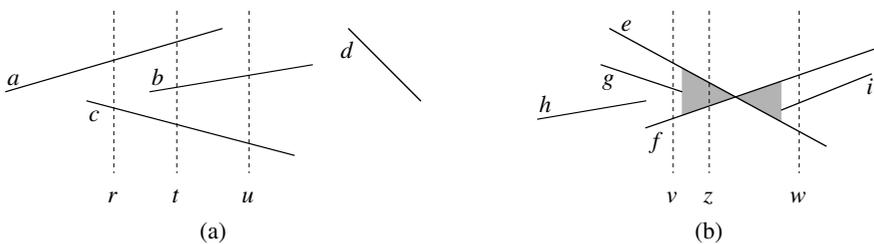


Figure 33.4 Le tri de segments de droite pour différentes droites de balayage verticales. (a) On a $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ et $b >_u c$. Le segment d n'est comparable avec aucun autre segment représenté. (b) Quand les segments e et f se coupent, leur ordre est inversé : on a $e >_v f$ mais $f >_w e$. Toute droite de balayage (tel z) qui traverse la région en gris a e et f consécutifs dans sa relation d'ordre total.

Pour tout x , la relation « $>_x$ » est une relation d'ordre total (voir section B.2) sur les segments qui coupent la droite de balayage en x . Cependant, l'ordre peut changer selon la valeur de x , au gré des entrées et des sorties des segments dans la liste de tri.

Un segment entre dans la liste de tri quand son extrémité gauche est atteinte par la droite, et il sort de la liste quand son extrémité droite est rencontrée par la droite de balayage.

Que se passe-t-il quand la droite de balayage traverse l'intersection de deux segments ? Comme le montre la figure 33.4(b), leurs positions dans l'ordre total sont inversées. Les droites de balayage v et w sont respectivement à gauche et à droite du point d'intersection des segments e et f , et l'on a $e >_v f$ et $f >_w e$. Notez que, grâce à notre hypothèse selon laquelle trois segments ne se coupent pas en un même point, il doit exister une droite de balayage verticale x pour lequel les segments sécants e et f sont *consécutifs* dans l'ordre total $>_x$. Pour toute droite de balayage, par exemple z , qui traverse la région en gris de la figure 33.4(b), e et f sont consécutifs dans sa relation d'ordre total.

b) Déplacement d'une droite de balayage

Les algorithmes de balayage gèrent en général deux ensembles de données :

- 1) L'*état de la droite de balayage* donne les relations entre les objets coupés par le faisceau.
- 2) L'*échancier des points d'événement* est une séquence d'abscisses, ordonnées de la gauche vers la droite, qui définit les positions d'arrêt de la droite de balayage. On appelle une telle position d'arrêt un *point d'événement*. Les modifications de l'état de la droite ne se produisent qu'aux points d'événement.

Pour certains algorithmes (l'algorithme demandé à l'exercice 33.2.7, par exemple), l'échancier des points d'événement est déterminé dynamiquement pendant la progression de l'algorithme. L'algorithme qui va suivre détermine les points d'événement de manière statique, en se basant uniquement sur des propriétés simples des données d'entrée. En particulier, chaque extrémité de segment est un point d'événement. On trie les extrémités des segments en augmentant l'abscisse et l'on procède de la gauche vers la droite. (Si deux ou plusieurs extrémités sont *coverticales*, c'est à dire si elles ont la même abscisse, alors on les découple en plaçant toutes les extrémités gauches couverticales avant les extrémités droites couverticales. Au sein d'un ensemble d'extrémités gauches couverticales, on commence par les ordonnées les plus petites de la droite ; et l'on fait de même au sein d'un ensemble d'extrémités droites couverticales.) On insère un segment dans l'état de la droite de balayage quand son extrémité gauche est rencontrée, et on le supprime de l'état quand son extrémité droite est rencontrée. Dès que deux segments deviennent consécutifs dans l'ordre total, on vérifie s'ils sont sécants.

L'état de la droite de balayage est un ordre total T , pour lequel on exige les opérations suivantes :

- INSÉRER(T, s) : insère le segment s dans T .
- SUPPRIMER(T, s) : supprime le segment s de T .

- $\text{AU-DESSUS}(T, s)$: retourne le segment qui suit immédiatement le segment s dans T .
- $\text{AU-DESSOUS}(T, s)$: retourne le segment qui précède immédiatement le segment s dans T .

Si l'on dispose de n segments en entrée, on peut effectuer chacune des opérations susmentionnées en temps $O(\lg n)$ en utilisant des arbres rouge-noir. Rappelez-vous que les opérations d'arbre rouge-noir vues au chapitre 13 impliquent des comparaisons de clés. On peut remplacer les comparaisons de clés par des comparaisons qui emploient des produits en croix pour déterminer l'ordre relatif de deux segments (voir exercice 33.2.2).

c) Pseudo code pour l'intersection des segments

L'algorithme ci-après prend en entrée un ensemble S de n segments de droite, puis retourne soit la valeur booléenne VRAI si deux segments quelconques de S se coupent, soit la valeur FAUX dans le cas contraire. L'ordre total T est implémenté par un arbre rouge-noir.

INTERSECTION-DEUX-SEGMENTS-QUELCONQUES(S)

```

1   $T \leftarrow \emptyset$ 
2  trier les extrémités des segments de  $S$  de la gauche vers la droite,
   si égalité, placer les extrémités gauches avant les extrémités droites
   puis, dans chacun de ces deux ensembles, trier dans l'ordre
   des ordonnées des points
3  pour tout point  $p$  de la liste triée des extrémités
4      faire si  $p$  est l'extrémité gauche d'un segment  $s$ 
5          alors INSÉRER( $T, s$ )
6          si ( $\text{AU-DESSUS}(T, s)$  existe et coupe  $s$ )
              ou ( $\text{AU-DESSOUS}(T, s)$  existe et coupe  $s$ )
7              alors retourner VRAI
8      si  $p$  est l'extrémité droite d'un segment  $s$ 
9          alors si  $\text{AU-DESSUS}(T, s)$  et  $\text{AU-DESSOUS}(T, s)$  existe
                  et  $\text{AU-DESSUS}(T, s)$  coupe  $\text{AU-DESSOUS}(T, s)$ 
10         alors retourner VRAI
11         SUPPRIMER( $T, s$ )
12 retourner FAUX

```

La figure 33.5 illustre l'exécution de l'algorithme. La ligne 1 initialise l'ordre total en lui affectant l'ensemble vide. La ligne 2 détermine l'échéancier des points d'événement en triant les $2n$ extrémités de segment de la gauche vers la droite (en cas d'égalité, on procède comme susmentionné). Notez que la ligne 2 peut être exécutée en triant lexicographiquement les extrémités sur (x, y) , où x et y sont les coordonnées usuelles et $e = 0$ pour une extrémité gauche et $e = 1$ pour une extrémité droite.

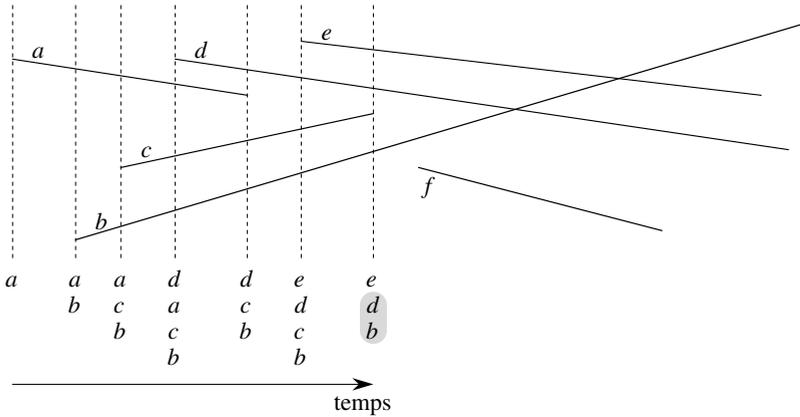


Figure 33.5 Exécution de la procédure INTERSECTION-DEUX-SEGMENTS-QUELCONQUES. Chaque ligne pointillée représente la droite de balayage en un point d'événement, et l'ordre des noms de segment sous chaque droite de balayage est l'ordre total T à la fin de la boucle **pour** dans laquelle le point d'événement correspondant est traité. L'intersection des segments d et b est trouvée quand le segment c est supprimé.

Chaque itération de la boucle **pour** des lignes 3–11 traite un seul point d'événement p . Si p est l'extrémité gauche d'un segment s , la ligne 5 ajoute s à l'ordre total et les lignes 6–7 retournent VRAI si s coupe l'un ou l'autre des segments qui lui sont consécutifs dans l'ordre total défini par la droite de balayage traversant p . (Une condition limite survient si p se trouve sur un autre segment s' . Dans ce cas, on exige seulement que s et s' soient placés consécutivement dans T .) Si p est l'extrémité droite d'un segment s , alors s doit être supprimé de l'ordre total. Les lignes 9–10 retournent VRAI s'il existe une intersection entre les segments qui encadrent s dans l'ordre total défini par la droite de balayage traversant p ; ces segments deviendront consécutifs dans l'ordre total quand s sera supprimé. Si ces segments ne se coupent pas, la ligne 11 supprime le segment s de l'ordre total. Enfin, si aucune intersection n'est trouvée pendant le traitement des $2n$ points d'événement, la ligne 12 retourne FAUX.

d) Validité

Pour montrer que INTERSECTION-DEUX-SEGMENTS- est conforme, nous allons prouver que l'appel à INTERSECTION-DEUX-SEGMENTS-QUELCONQUES(S) retourne VRAI si et seulement si il existe une intersection parmi les segments de S .

Il est facile de voir que INTERSECTION-DEUX-SEGMENTS-QUELCONQUES ne retourne VRAI (lignes 7 et 10) que si elle trouve une intersection entre deux des segments donnés en entrée. Donc, si elle retourne VRAI, c'est qu'il y a une intersection.

Il faut aussi démontrer la réciproque : s'il y a une intersection, alors INTERSECTION-DEUX-SEGMENTS-QUELCONQUES retourne VRAI. Supposons qu'il y ait au moins une intersection. Soit p le point d'intersection le plus à gauche (en cas d'égalité,

on choisit celui ayant la plus petite ordonnée) et soient a et b les segments qui sont sécants en p . Comme aucune intersection n'a lieu à gauche de p , l'ordre donné par T est correct pour tous les points à gauche de p . Comme trois segments ne peuvent pas se couper en un même point, il existe une droite de balayage z pour lequel a et b deviennent consécutifs dans l'ordre total ⁽²⁾. Par ailleurs, z se trouve à gauche de p ou traverse p . Il existe une extrémité de segment q sur la droite de balayage z qui est le point d'événement où a et b deviennent consécutifs dans l'ordre total. Si p est sur la droite de balayage z , alors $q = p$. Si p n'est pas sur la droite de balayage z , alors q est à gauche de p . Dans les deux cas, l'ordre imposé par T est correct juste avant que q soit rencontré. (Ici, on s'appuie sur le fait que p est le plus bas des points d'intersection les plus à gauche. A cause de l'ordre lexicographique dans lequel les points d'événement sont traités, même si p se trouve sur la droite de balayage z et qu'il existe un autre point d'intersection p' sur z , le point d'événement $q = p$ est traité avant que p' puisse interférer avec l'ordre total T . En outre, même si p est l'extrémité gauche d'un segment, par exemple a , et l'extrémité droite de l'autre segment, par exemple b , comme les événements d'extrémité gauche se produisent avant les événements d'extrémité droite, le segment b est dans T quand la droite de balayage rencontre le segment a .) Soit le point d'événement q est traité par INTERSECTION-DEUX-SEGMENTS-QUELCONQUES, soit il ne l'est pas.

Si q est traité par INTERSECTION-DEUX-SEGMENTS-QUELCONQUES, il n'y a que deux possibilités d'action :

- 1) a ou b est inséré dans T , et l'autre segment se trouve au-dessus ou au-dessous de lui dans l'ordre total. Les lignes 4–7 détectent ce cas.
- 2) Les segments a et b sont déjà dans T , et un segment situé entre eux dans l'ordre total est supprimé, ce qui les rend consécutifs. Les lignes 8–11 détectent ce cas.

Dans un cas comme dans l'autre, l'intersection p est trouvée et INTERSECTION-DEUX-SEGMENTS-QUELCONQUES retourne VRAI.

Si le point d'événement q n'est pas traité par INTERSECTION-DEUX-SEGMENTS-QUELCONQUES, c'est que la procédure a rendu la main avant de traiter tous les points d'événement. Cette situation ne pourrait advenir que si INTERSECTION-DEUX-SEGMENTS-QUELCONQUES a déjà trouvé une intersection et retourné VRAI.

Donc, s'il y a une intersection, INTERSECTION-DEUX-SEGMENTS-QUELCONQUES retourne VRAI. Comme nous l'avons déjà vu, si INTERSECTION-DEUX-SEGMENTS-QUELCONQUES retourne VRAI, c'est qu'il y a une intersection. Par conséquent, INTERSECTION-DEUX-SEGMENTS-QUELCONQUES retourne toujours une réponse correcte.

(2) Si l'on autorisait trois segments à se croiser en un même point, il pourrait y avoir un segment c qui couperait a et b au point p . Autrement dit, on pourrait avoir $a <_w c$ et $c <_w b$ pour toute droite de balayage w situé à gauche de p pour lequel $a <_w b$. L'exercice 33.2.8 vous demandera de montrer que INTERSECTION-DEUX-SEGMENTS-QUELCONQUES est correcte même s'il existe trois segments qui se coupent en un même point.

e) Temps d'exécution

S'il existe n segments dans l'ensemble S , alors la procédure INTERSECTION-DEUX-SEGMENTS-QUELCONQUES s'exécute en temps $O(n \lg n)$. La ligne 1 prend un temps $O(1)$. La ligne 2 prend un temps $O(n \lg n)$, en utilisant le tri par fusion ou le tri par tas. Puisqu'il existe $2n$ points d'événement, la boucle **pour** des lignes 3–11 se répète au plus $2n$ fois. Chaque itération prend un temps $O(\lg n)$, car chaque opération d'arbre rouge-noir demande un temps $O(\lg n)$ et que, avec la méthode de la section 33.1, chaque test d'intersection prend un temps $O(1)$. Le temps total d'exécution est donc $O(n \lg n)$.

Exercices

33.2.1 Montrer qu'il peut exister $\Theta(n^2)$ intersections dans un ensemble de n segments de droite.

33.2.2 Étant donnés deux segments a et b qui sont comparables en x , montrer comment déterminer en temps $O(1)$ laquelle des deux propriétés $a >_x b$ ou $b >_x a$ est vraie. On supposera qu'aucun des deux segments n'est vertical. (*conseil* : Si a et b ne se coupent pas, vous pouvez vous contenter d'utiliser des produits en croix. Si a et b se coupent, ce que vous pouvez naturellement déterminer en n'utilisant que des produits en croix, vous pouvez encore vous contenter d'utiliser l'addition, la soustraction et la multiplication en évitant la division. Bien évidemment, dans l'application de la relation $>_x$ utilisée ici, si a et b se coupent, on peut tout simplement stopper et déclarer qu'on a trouvé une intersection.)

33.2.3 Le professeur Maginot suggère qu'on modifie la procédure INTERSECTION-DEUX-SEGMENTS-QUELCONQUES pour que, au lieu de s'arrêter après avoir trouvé une intersection, elle imprime les segments sécants et passe à l'itération suivante de la boucle **pour**. Le professeur appelle la procédure résultante IMPRIMER-SEGMENTS-SÉCANTS et affirme qu'elle imprime toutes les intersections, de la gauche vers la droite, à mesure qu'elles apparaissent dans l'ensemble des segments de droite. Montrer que le professeur se trompe à deux reprises, en exhibant un ensemble de segments pour lequel la première intersection trouvée par IMPRIMER-SEGMENTS-SÉCANTS n'est pas la plus à gauche et un ensemble de segments pour lequel IMPRIMER-SEGMENTS-SÉCANTS n'arrive pas à trouver toutes les intersections.

33.2.4 Donner un algorithme en temps $O(n \lg n)$ qui détermine si un polygone à n sommets est simple.

33.2.5 Donner un algorithme en temps $O(n \lg n)$ qui détermine si deux polygones simples ayant au total n sommets se coupent.

33.2.6 Un *disque* est composé d'un cercle et de l'intérieur du cercle ; on le représente par son point central et un rayon. Deux disques se recouvrent s'ils possèdent un point commun. Donner un algorithme en $O(n \lg n)$ pour déterminer si deux disques quelconques d'un ensemble de n disques se coupent.

33.2.7 Étant donné un ensemble de n segments de droite contenant un total de k intersections, montrer comment imprimer les k intersections en temps $O((n+k) \lg n)$.

33.2.8 Prouver que INTERSECTION-DEUX-SEGMENTS-QUELCONQUES fonctionne correctement même s'il y a trois ou plusieurs segments qui se rencontrent en un même point.

33.2.9 Montrer que INTERSECTION-DEUX-SEGMENTS-QUELCONQUES est correcte même en présence de segments verticaux, à condition que l'extrémité inférieure d'un segment vertical soit traitée comme si c'était une extrémité gauche et que l'extrémité supérieure soit traitée comme si c'était une extrémité droite. Comment est modifiée votre réponse à l'exercice 33.2.2, si l'on autorise les segments verticaux ?

33.3 RECHERCHE DE L'ENVELOPPE CONVEXE

L'enveloppe convexe d'un ensemble Q de points est le plus petit polygone convexe P tel que chaque point de Q est soit sur le contour de P , soit à l'intérieur. (Voir exercice 33.1.5 pour la définition exacte d'un polygone convexe.) L'enveloppe convexe de Q est notée $EC(Q)$. Intuitivement, on peut se représenter les points de Q comme autant d'épingles plantées sur un tableau. L'enveloppe convexe est alors le contour formé par un élastique serré qui entoure toutes les épingles. La figure 33.6 montre un ensemble de points et son enveloppe convexe.

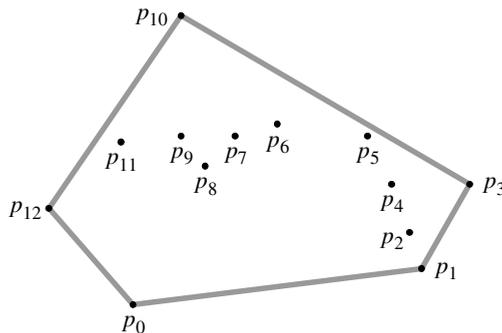


Figure 33.6 Un ensemble de points Q avec son enveloppe convexe $EC(Q)$ en gris.

Dans cette section, nous allons présenter deux algorithmes capables de calculer l'enveloppe convexe d'un ensemble de n points. Les deux algorithmes affichent les sommets de l'enveloppe convexe dans l'ordre contraire des aiguilles d'une montre. Le premier, dit balayage de Graham, s'exécute en temps $O(n \lg n)$. Le second, dit parcours de Jarvis, s'exécute en temps $O(nh)$, où h est le nombre de sommets de l'enveloppe convexe. Comme on peut le voir sur la figure 33.6, chaque sommet de $EC(Q)$ est un point de Q . Les deux algorithmes exploitent cette propriété, en décidant

quels sont les sommets dans Q qu'il faut conserver comme sommets de l'enveloppe convexe et quels sont ceux qu'il faut rejeter.

Il existe en fait plusieurs moyens de calculer les enveloppes convexes en temps $O(n \lg n)$. Le balayage de Graham comme le parcours de Jarvis utilisent une technique appelée « balayage circulaire », qui traite les sommets dans l'ordre des angles polaires qu'ils forment avec un sommet de référence. Parmi les autres méthodes, on peut citer :

- La **méthode incrémentielle**, où les points sont triés de gauche à droite, ce qui génère une séquence $\langle p_1, p_2, \dots, p_n \rangle$. A la i ème étape, l'enveloppe convexe des $i - 1$ points les plus à gauche, $EC(\{p_1, p_2, \dots, p_{i-1}\})$, est mise à jour en fonction du i ème point à partir de la gauche, pour former $EC(\{p_1, p_2, \dots, p_i\})$. Comme cela vous sera demandé à l'exercice 33.3.6, cette méthode peut être implémentée pour prendre un temps global $O(n \lg n)$.
- La **méthode diviser-pour-régner**, à temps $\Theta(n)$, où l'ensemble des n points est divisé en deux sous-ensembles, l'un contenant les $\lceil n/2 \rceil$ points les plus à gauche et l'autre les $\lfloor n/2 \rfloor$ points les plus à droite. Les enveloppes convexes des sous-ensembles sont calculées récursivement, puis une technique astucieuse les regroupe en un temps $O(n)$. Le temps d'exécution est décrit par la récurrence bien connue $T(n) = 2T(n/2) + O(n)$, ce qui fait que la méthode diviser-pour-régner tourne en temps $O(n \lg n)$.
- La **méthode par greffes successives**, qui est similaire à l'algorithme à temps linéaire (dans le cas le plus défavorable) du médian, vu à la section 9.3. Elle trouve la portion supérieure (ou « chaîne supérieure ») de l'enveloppe convexe en éliminant de façon répétée une fraction constante des points restants, jusqu'à ne conserver que la chaîne supérieure de l'enveloppe convexe. Le même traitement est ensuite appliqué à la chaîne inférieure. Cette méthode est la plus rapide asymptotiquement : si l'enveloppe convexe contient h sommets, elle s'exécute en temps $O(n \lg h)$ seulement.

Le calcul de l'enveloppe convexe d'un ensemble de points est un problème intéressant en lui-même. Par ailleurs, certains algorithmes conçus pour d'autres problèmes de géométrie algorithmique commencent par calculer une enveloppe convexe. Considérons, par exemple, le problème à deux dimensions des **paires de points les plus éloignés** : on se donne un ensemble de n points du plan et on souhaite trouver les deux points dont la distance de l'un à l'autre est maximale. Comme l'exercice 33.3.3 vous demandera de le montrer, ces deux points doivent être des sommets de l'enveloppe convexe. Bien que nous n'en fassions pas la démonstration ici, la paire de sommets les plus éloignés d'un polygone convexe à n sommets peut être trouvée en temps $O(n)$. Donc, en calculant l'enveloppe convexe des n points d'entrée en $O(n \lg n)$ puis en trouvant la paire de points les plus éloignés mutuellement parmi les sommets du polygone convexe résultant, on peut trouver en temps $O(n \lg n)$ la paire de points les plus éloignés mutuellement dans un ensemble de n points.

a) Balayage de Graham

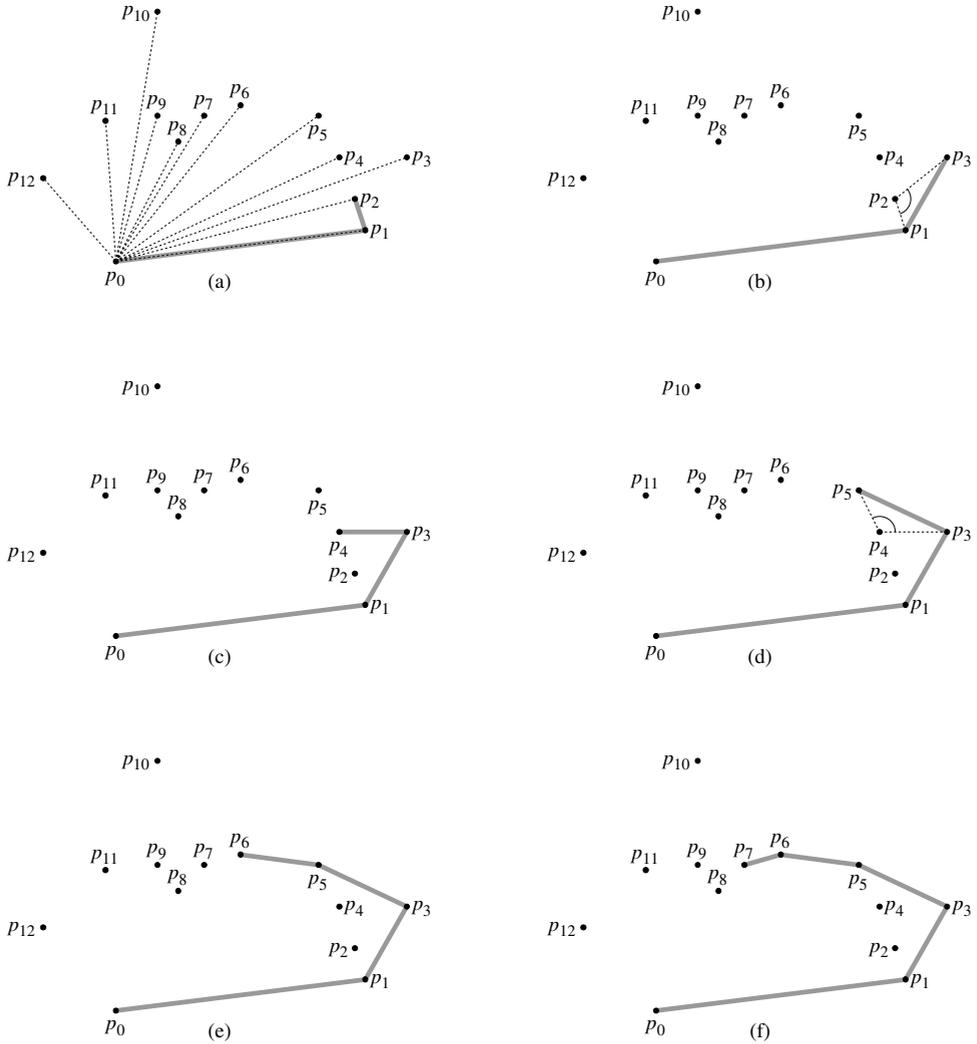
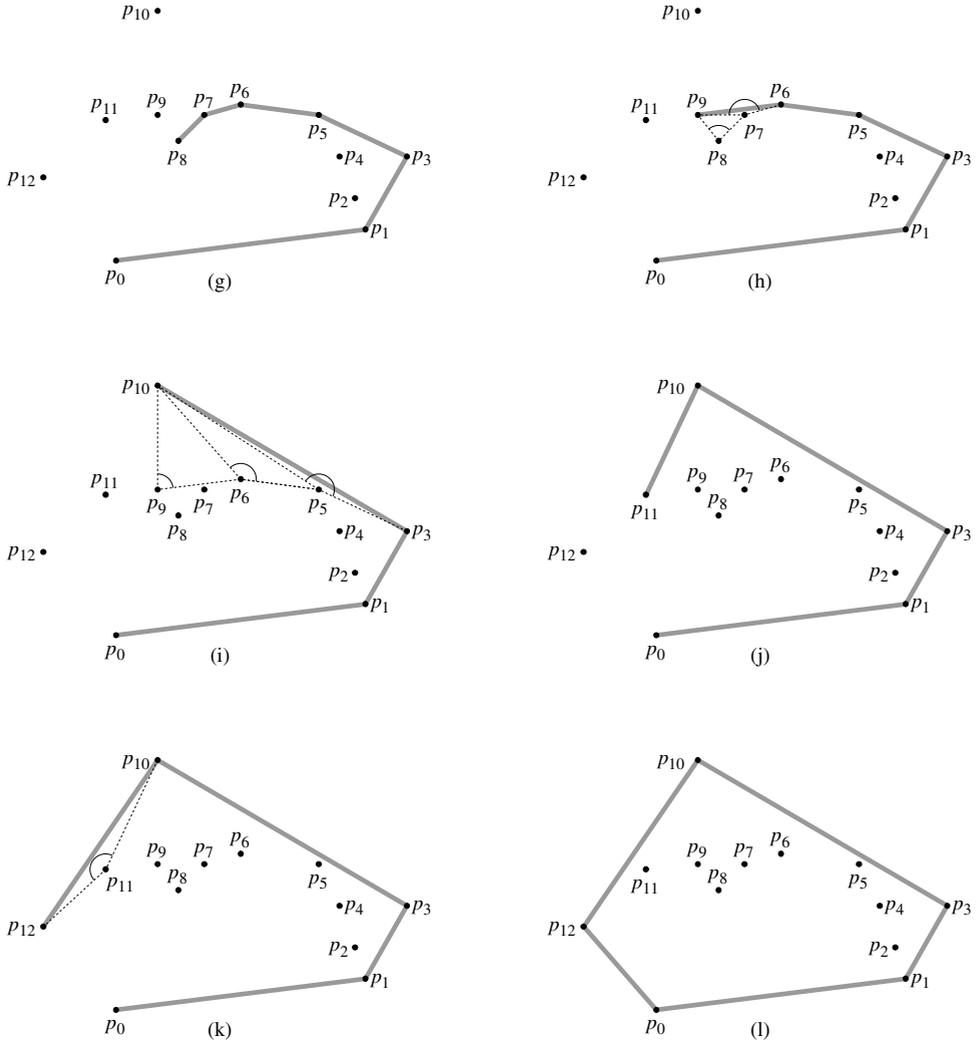


Figure 33.7 L'exécution de BALAYAGE-GRAHAM sur l'ensemble Q de la figure 33.6. L'enveloppe convexe courante contenue dans la pile S apparaît en gris à chaque étape. (a) La suite $\langle p_1, p_2, \dots, p_{12} \rangle$ de points, numérotés dans l'ordre croissant d'angle polaire relativement à p_0 , et la pile initiale S contenant p_0, p_1 et p_2 . (b)–(k) La pile S après chaque itération de la boucle **pour** des lignes 6–9. Les lignes pointillées montrent les bifurcations qui ne sont pas dirigées vers la gauche, ce qui a pour effet de dépiler les points. A la partie (h), par exemple, la bifurcation à droite au niveau de l'angle $\angle p_7 p_8 p_9$ dépile p_8 puis la bifurcation à droite au niveau de l'angle $\angle p_6 p_7 p_9$ dépile p_7 . (l) L'enveloppe convexe retournée par la procédure correspond à celle de la figure 33.6.



Le **balayage de Graham** résout le problème de l’enveloppe convexe en gérant une pile S de points candidats. Chaque point de l’ensemble Q est empilé une fois, puis les points qui ne sont pas des sommets de $EC(Q)$ finissent par être tous dépilés. Quand l’algorithme se termine, la pile S contient exactement les sommets de $EC(Q)$, dans l’ordre trigonométrique de leur apparition sur le contour.

La procédure BALAYAGE-GRAHAM prend en entrée un ensemble Q de points, où $|Q| \geq 3$. Elle appelle la fonction SOMMET(S), qui retourne le point situé au sommet de la pile S sans changer S et la fonction SOUS-SOMMET(S), qui retourne le point situé juste en-dessous du sommet de la pile S sans changer S . Comme nous le verrons bientôt, la pile S retournée par BALAYAGE-GRAHAM contient, du bas vers le haut, exactement les sommets de $EC(Q)$ dans l’ordre trigonométrique.

Lorsqu'on parcourt l'enveloppe convexe dans le sens trigonométrique, on doit bifurquer à gauche au niveau de chaque sommet. Donc, chaque fois que la boucle **tant que** trouve un sommet pour lequel la bifurcation ne se fait pas vers la gauche, ce sommet est dépilé. (En testant les bifurcations qui ne se font pas vers la gauche et non pas les bifurcations qui se font vers la droite, on exclut le cas d'un angle plat en un sommet de l'enveloppe convexe résultante. En effet, aucun sommet d'un polygone convexe ne doit être une combinaison convexe d'autres sommets du polygone.) Après avoir dépilé tous les sommets pour lesquels la bifurcation ne se fait pas vers la gauche quand on se dirige vers le point p_i , on empile p_i . Les figures 33.7(b)–(k) montrent l'état de la pile S après chaque itération de la boucle **pour**. Au final, BALAYAGE-GRAHAM retourne la pile S en ligne 10. La figure 33.7(l) montre l'enveloppe convexe correspondante.

Le théorème suivant valide formellement la procédure BALAYAGE-GRAHAM.

Théorème 33.1 (Conformité du balayage de Graham) *Si BALAYAGE-GRAHAM est exécutée sur un ensemble Q de points tel que $|Q| \geq 3$, alors, à la fin de la procédure, la pile S contient, du bas vers le haut, les sommets de $EC(Q)$ pris dans l'ordre inverse des aiguilles d'une montre.*

Démonstration : Après la ligne 2, on a la suite de points $\langle p_1, p_2, \dots, p_m \rangle$. Définissons, pour $i = 2, 3, \dots, m$, le sous-ensemble de points $Q_i = \{p_0, p_1, \dots, p_i\}$. Les points de $Q - Q_m$ sont ceux qui ont été supprimés parce qu'ils avaient le même angle polaire par rapport à p_0 qu'un certain point de Q_m ; ces points ne sont pas dans $EC(Q)$, et donc $EC(Q_m) = EC(Q)$. Par conséquent, il suffit de montrer que, quand BALAYAGE-GRAHAM se termine, la pile S se compose des sommets de $EC(Q_m)$ pris dans l'ordre inverse des aiguilles d'une montre (en partant du bas de la pile). Notez que, de même que p_0, p_1 et p_m sont des sommets de $EC(Q)$, les points p_0, p_1 et p_i sont tous des sommets de $EC(Q_i)$.

La démonstration repose sur l'invariant de boucle suivant : Au début de chaque itération de la boucle **pour** des lignes 6–9, la pile S contient, en partant du bas et en allant vers le haut, les sommets de $EC(Q_{i-1})$ pris dans l'ordre inverse des aiguilles d'une montre.

Initialisation : L'invariant est vrai la première fois que l'on exécute la ligne 6 ; en effet, à ce moment-là, la pile S contient exactement les sommets de $Q_2 = Q_{i-1}$, ensemble de trois sommets qui est sa propre enveloppe convexe. En outre, ces sommets apparaissent dans l'ordre inverse des aiguilles d'une montre quand on parcourt la pile du bas vers le haut.

Conservation : Quand on attaque une itération de la boucle **pour**, le sommet de la pile S est p_{i-1} , qui avait été empilé à la fin de la précédente itération (ou avant la première itération si $i = 3$). Soit p_j le sommet de S après exécution de la boucle **tant que** des lignes 7–8 mais avant que la ligne 9 empile p_i , et soit p_k le point juste au-dessous de p_j dans S . Au moment où p_j est le sommet de S et que p_i n'a pas encore été empilé, la pile S contient exactement les mêmes points qu'elle contenait après l'itération j de la boucle **pour**. D'après l'invariant, S contient donc exactement les sommets de $EC(Q_j)$ à cet instant ; ces sommets apparaissent dans l'ordre inverse des aiguilles d'une montre, quand on parcourt la pile du bas vers le haut.

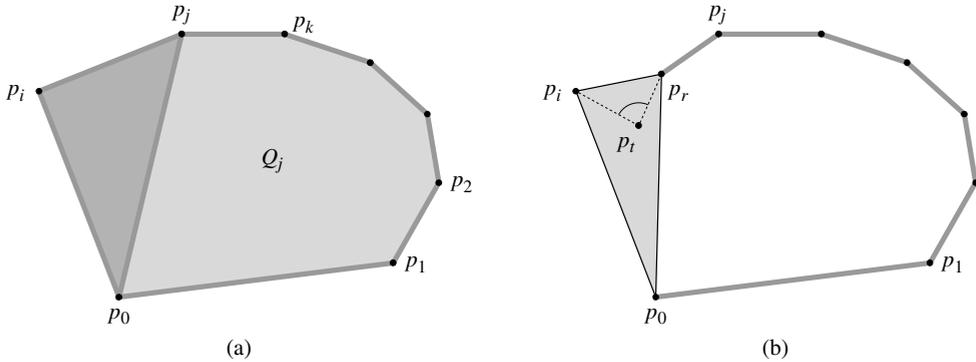


Figure 33.8 Démonstration de la validité de BALAYAGE-GRAHAM. **(a)** Comme l'angle polaire de p_i relativement à p_0 est plus grand que l'angle polaire de p_j , et comme l'angle $\angle p_k p_j p_i$ tourne vers la gauche, l'ajout de p_i à $EC(Q_j)$ donne exactement les sommets de $EC(Q_j \cup \{p_i\})$. **(b)** Si l'angle $\angle p_r p_i p_j$ ne tourne pas vers la gauche, alors p_t est soit à l'intérieur du triangle formé par p_0 , p_r et p_i , soit sur un côté du triangle, et il ne peut pas être un sommet de $EC(Q_i)$.

Continuons à nous intéresser à ce qui se passe juste avant que p_i soit empilé. En se référant à la figure 33.8(a) (l'angle polaire de p_i relativement à p_0 est supérieur à l'angle polaire de p_j), comme l'angle $\angle p_k p_j p_i$ tourne vers la gauche (autrement, on aurait dépilé p_j), on voit que, puisque S contient exactement les sommets de $EC(Q_j)$, une fois qu'on a empilé p_i , la pile S contient exactement les sommets de $EC(Q_j \cup \{p_i\})$, toujours pris dans l'ordre inverse des aiguilles d'une montre (quand on remonte la pile).

Nous allons montrer maintenant que $EC(Q_j \cup \{p_i\})$ est le même ensemble de points que $EC(Q_i)$. Soit un point p_t , quelconque ayant été dépilé lors de l'itération i de la boucle **pour**, et soit p_r le point qui était juste au-dessous de p_t dans la pile S au moment où p_t a été dépilé (p_r pourrait être éventuellement p_j). L'angle $\angle p_r p_i p_j$ ne tourne pas vers la gauche et l'angle polaire de p_t relativement à p_0 est supérieur à l'angle polaire de p_r . Comme le montre la figure 33.8(b), p_t est forcément soit à l'intérieur du triangle formé par p_0 , p_r et p_i , soit sur un côté de ce triangle (mais ce n'est pas un sommet du triangle). p_t étant dans un triangle formé par trois autres points de Q_i , il ne peut visiblement pas être un sommet de $EC(Q_i)$. Comme p_t n'est pas un sommet de $EC(Q_i)$, on a

$$EC(Q_i - \{p_t\}) = EC(Q_i) . \tag{33.1}$$

Soit P_i l'ensemble des points ayant été dépilés lors de l'itération i de la boucle **pour**. Comme l'égalité (33.1) s'applique à tous les points de P_i , on peut l'appliquer de manière répétée pour montrer que $EC(Q_i - P_i) = EC(Q_i)$. Mais $Q_i - P_i = Q_j \cup \{p_i\}$, et on conclut donc que $EC(Q_j \cup \{p_i\}) = EC(Q_i - P_i) = EC(Q_i)$. Nous avons montré que, une fois que nous avons empilé p_i , la pile S contient exactement les sommets de $EC(Q_i)$, pris dans l'ordre inverse des aiguilles d'une montre quand on parcourt la pile du bas vers le haut. Incrémenter i a donc pour effet de conserver l'invariant pour la prochaine itération.

Terminaison : Quand la boucle se termine, on a $i = m + 1$; et donc l'invariant implique que la pile S contient exactement les sommets de $EC(Q_m)$, qui n'est autre que $EC(Q)$, pris dans l'ordre inverse des aiguilles d'une montre (quand on remonte la pile). Cela parachève la démonstration. \square

Montrons à présent que le temps d'exécution de BALAYAGE-GRAHAM est $O(n \lg n)$, où $n = |Q|$. La ligne 1 prend un temps $\Theta(n)$. La ligne 2 prend un temps $O(n \lg n)$, si elle utilise le tri par fusion ou par tas pour trier les angles polaires plus la méthode des produits en croix décrite à la section 33.1 pour comparer les angles. (La suppression de tous les points de même angle polaire hormis le plus éloigné peut s'effectuer en temps $O(n)$ au total.) Les lignes 3–5 prennent un temps $O(1)$. Comme $m \leq n - 1$, la boucle **pour** des lignes 6–9 est exécutée au plus $n - 3$ fois. Comme EMPILER prend un temps $O(1)$, chaque itération prend un temps $O(1)$ si l'on ne compte pas le temps passé dans la boucle **tant que** des lignes 7–8 ; donc, globalement, la boucle **pour** prend un temps $O(n)$ non compris la boucle **tant que** imbriquée.

On va utiliser la méthode par agrégat pour montrer que la boucle **tant que** nécessite au total un temps $O(n)$. Pour $i = 0, 1, \dots, m$, chaque point p_i est empilé sur S exactement une fois. Comme pour l'analyse de la procédure MULTIEMPILE de la section 17.1, observons qu'il existe au plus une opération DÉPILER pour chaque opération EMPILER. Il existe au moins trois points (p_0, p_1 et p_m) qui ne sont jamais dépilés, et on effectue donc un total d'au plus $m - 2$ opérations DÉPILER. Chaque itération de la boucle **tant que** effectue un DÉPILER et il y a donc au total au plus $m - 2$ itérations de la boucle **tant que**. Comme le test de la ligne 7 prend un temps $O(1)$, chaque appel à DÉPILER prend un temps $O(1)$, et comme $m \leq n - 1$, le temps total requis pour la boucle **tant que** est $O(n)$. Le temps d'exécution de BALAYAGE-GRAHAM est donc $O(n \lg n)$.

b) Parcours de Jarvis

Le *parcours de Jarvis* calcule l'enveloppe convexe d'un ensemble Q de points par une technique dite du *paquet cadeau*. L'algorithme s'exécute en temps $O(nh)$, où h est le nombre de sommets de $EC(Q)$. Quand h est $o(\lg n)$, le parcours de Jarvis est plus rapide asymptotiquement que le balayage de Graham.

Intuitivement, le parcours de Jarvis simule l'emballage de l'ensemble Q par une feuille de papier tendue. On commence par attacher l'extrémité du papier au point le plus bas de l'ensemble, c'est-à-dire au même point p_0 d'où nous avons fait commencer le balayage de Graham. Ce point est un sommet de l'enveloppe convexe. On tire le papier vers la droite pour le tendre, puis on le tire vers le haut jusqu'à ce qu'il atteigne un point. Ce point doit aussi être un sommet de l'enveloppe convexe. En gardant le papier tendu, on continue de cette façon autour de l'ensemble des sommets, jusqu'à revenir au point de départ p_0 .

Plus formellement, le parcours de Jarvis construit une séquence $H = \langle p_0, p_1, \dots, p_{h-1} \rangle$ de sommets de $EC(Q)$. On commence à p_0 . Comme le montre la figure 33.9, le sommet suivant de l'enveloppe convexe p_1 a le plus petit angle polaire par rapport à p_0 . (En cas d'égalité, on choisit le point le plus éloigné de p_0 .) De même, p_2 aura le plus petit angle polaire par rapport à p_1 etc. Lorsqu'on atteint le plus haut sommet, disons p_k (en cas d'égalité, on choisit le plus éloigné de ces sommets), on a construit, comme le montre la figure 33.9, la **chaîne droite** de $EC(Q)$. Pour construire la **chaîne gauche**, on commence en p_k et on choisit pour p_{k+1} le point de plus petit angle polaire par rapport à p_k , mais *en partant du demi-axe des abscisses négatives*. On continue ainsi, formant la chaîne gauche en prenant des angles polaires exprimés à partir des abscisses négatives, jusqu'à revenir au sommet initial p_0 .

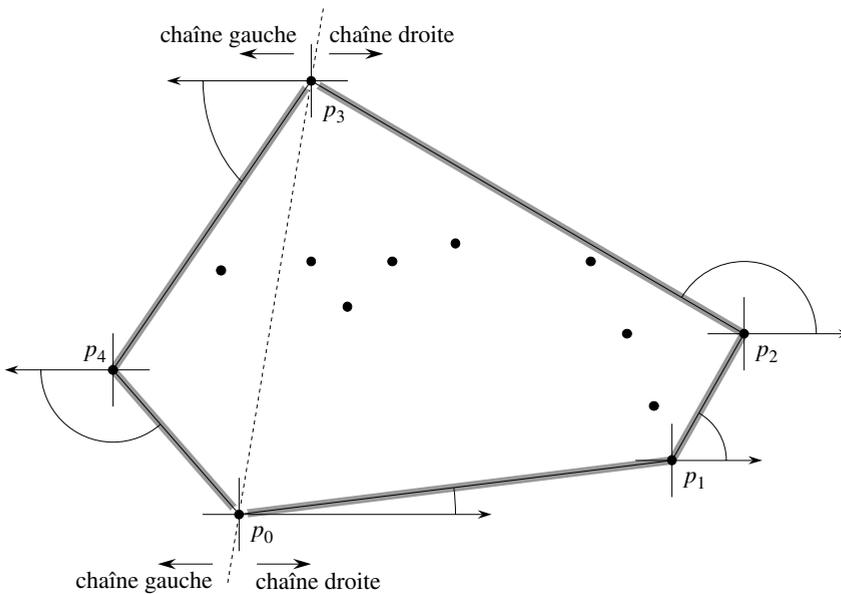


Figure 33.9 L'action du parcours de Jarvis. Le premier sommet choisi est le point le plus bas, p_0 . Le sommet suivant, p_1 , est celui qui a le plus petit angle polaire par rapport à p_0 . Ensuite, p_2 a le plus petit angle polaire par rapport à p_1 . La chaîne droite remonte jusqu'au plus haut point p_3 . Ensuite, la chaîne gauche est construite en trouvant les plus petits angles polaires par rapport aux x négatifs.

On pourrait implémenter le parcours de Jarvis en un seul balayage autour de l'enveloppe convexe, c'est-à-dire sans séparer la construction des chaînes droite et gauche. Ce type d'implémentation gère en général l'angle du dernier côté choisi de l'enveloppe convexe et impose que la séquence des angles des côtés de l'enveloppe soit strictement croissante (dans l'intervalle $[0, 2\pi]$). L'avantage d'une construction séparée des chaînes est qu'il est inutile de calculer les angles explicitement ; les techniques de la section 33.1 suffisent pour comparer les angles.

Bien implémenté, le parcours de Jarvis a un temps d'exécution $O(nh)$. Pour chacun des h sommets de $EC(Q)$, on trouve le sommet d'angle polaire minimal. Chaque comparaison entre angles polaires demande un temps $O(1)$ si l'on utilise les techniques de la section 33.1. Comme le montre la section 9.1, il est possible de calculer le minimum de n valeurs en temps $O(n)$ si chaque comparaison se fait en temps $O(1)$. Donc, le parcours de Jarvis requiert un temps $O(nh)$.

Exercices

33.3.1 Démontrer que, dans la procédure BALAYAGE-GRAHAM, les points p_1 et p_m doivent être des sommets de $EC(Q)$.

33.3.2 Considérons un modèle de calcul qui supporte l'addition, la comparaison et la multiplication, et pour lequel il existe une borne inférieure de $\Omega(n \lg n)$ pour trier n nombres. Démontrer que, dans un tel modèle, $\Omega(n \lg n)$ est une borne inférieure pour le calcul, dans l'ordre, des sommets de l'enveloppe convexe d'un ensemble de n points.

33.3.3 Étant donné un ensemble de points Q , démontrer que les deux points les plus éloignés l'un de l'autre doivent être des sommets de $EC(Q)$.

33.3.4 Pour un polygone P donné et un point q sur son contour, l'*ombre* de q est l'ensemble des points r tels que le segment \overline{qr} se trouve entièrement sur le contour ou à l'intérieur de P . Un polygone P est *en forme d'étoile* s'il existe un point p à l'intérieur de P qui se trouve dans l'ombre de tout point du contour de P . L'ensemble de tous les points p de ce type s'appelle le *noyau* de P . (Voir figure 33.10.) Étant donné un polygone en forme d'étoile P à n sommets, spécifié par ses sommets pris dans l'ordre inverse des aiguilles d'une montre, montrer comment calculer $EC(P)$ en temps $O(n)$.

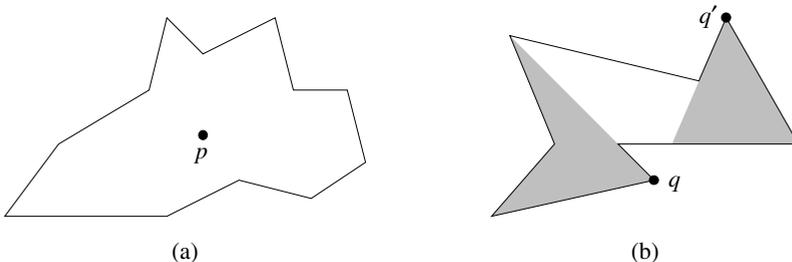


Figure 33.10 La définition d'un polygone en forme d'étoile, utilisée dans l'exercice 33.3.4. (a) Un polygone en forme d'étoile. Le segment reliant le point p à un point q quelconque du contour coupe le contour uniquement en q . (b) Un polygone qui n'est pas en forme d'étoile. La région grise de gauche est l'ombre de q et la région grise de droite est l'ombre de q' . Ces régions étant disjointes, le noyau est vide.

33.3.5 Dans le *problème de l'enveloppe convexe dynamique*, on a un ensemble Q de n points, qui sont donnés un par un. Après obtention de chaque point, il faut calculer l'enveloppe convexe des points obtenus jusqu'ici. Manifestement, on pourrait exécuter le balayage de Graham une fois pour chaque point, avec un temps total de $O(n^2 \lg n)$. Montrer que ce problème peut être résolu en temps $O(n^2)$ au total.

33.3.6 * Montrer comment implémenter la méthode incrémentielle pour calculer l'enveloppe convexe de n points de telle sorte qu'elle s'exécute en temps $O(n \lg n)$.

33.4 RECHERCHE DES DEUX POINTS LES PLUS RAPPROCHÉS

On considère maintenant le problème consistant à trouver les deux points les plus rapprochés dans un ensemble Q de $n \geq 2$ points. « Rapprochés » se réfère à la distance euclidienne habituelle : la distance entre les points $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$ est $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Deux points de l'ensemble Q peuvent coïncider, auquel cas la distance entre eux vaut zéro. Ce problème a notamment des applications dans les systèmes de contrôle de trafic. Un contrôleur de trafic aérien ou maritime peut avoir besoin de savoir quels sont les deux appareils les plus proches mutuellement pour détecter des collisions potentielles.

Un algorithme primaire se contenterait de tester les $\binom{n}{2} = \Theta(n^2)$ paires de points. Dans cette section, nous allons décrire un algorithme diviser-pour-régner dont le temps d'exécution est décrit par la récurrence familière $T(n) = 2T(n/2) + O(n)$. Donc, cet algorithme requiert un temps $O(n \lg n)$ seulement.

a) Algorithme diviser-pour-régner

Chaque invocation récursive de l'algorithme prend en entrée un sous-ensemble $P \subseteq Q$ et des tableaux X et Y , contenant chacun tous les points du sous-ensemble d'entrée P . Les points du tableau X sont triés par ordre monotone croissant des abscisses. De même, le tableau Y est trié par ordre monotone croissant des ordonnées. Notez que, pour atteindre la borne $O(n \lg n)$, on ne peut pas se permettre de trier dans chaque appel récursif ; si on le faisait, la récurrence du temps d'exécution serait $T(n) = 2T(n/2) + O(n \lg n)$, dont la solution est $T(n) = O(n \lg^2 n)$. (Utilisez la version de la méthode générale donnée à l'exercice 4.4.2). Nous verrons un peu plus loin comment utiliser le « pré tri » pour gérer cette propriété de tri sans réellement trier dans chaque appel récursif.

Un appel récursif donné, avec les entrées P , X et Y , commence par tester si $|P| \leq 3$. Dans ce cas, l'appel se contente d'appliquer la méthode primaire précédemment décrite : les $\binom{|P|}{2}$ paires de points sont toutes étudiées l'une après l'autre et on retourne les deux points les plus rapprochés. Si $|P| > 3$, l'appel récursif implémente le paradigme diviser-pour-régner de la façon suivante.

Diviser : La procédure trouve une droite verticale l qui sépare l'ensemble de points P en deux ensembles P_G et P_D tels que $|P_G| = \lceil |P|/2 \rceil$, $|P_D| = \lfloor |P|/2 \rfloor$, tous les points de P_G sont sur ou à gauche de la droite l , et tous les points de P_D sont sur ou à droite de la droite l . Le tableau X est divisé en deux tableaux X_G et X_D , qui contiennent respectivement les points de P_G et P_D , triés par ordre monotone croissant des abscisses. De même, le tableau Y est divisé en deux tableaux Y_G et Y_D , qui contiennent respectivement les points de P_G et P_D , triés par ordre monotone croissant des ordonnées.

Conquérir : Ayant divisé P en P_G et P_D , la procédure effectue deux appels récursifs, l'un pour trouver les deux points les plus rapprochés dans P_G et l'autre pour trouver les deux points les plus rapprochés dans P_D . Les entrées du premier appel sont le sous-ensemble P_G et les tableaux X_G et Y_G ; le second appel reçoit les entrées P_D , X_D et Y_D . Soient δ_G et δ_D les plus petites distances respectivement retournées pour P_G et P_D , et soit $\delta = \min(\delta_G, \delta_D)$.

Combiner : La paire de points les plus rapprochés est soit la paire ayant la distance δ trouvée par l'un des appels récursifs, soit une paire de points dont l'un est dans P_G et l'autre dans P_D . L'algorithme détermine s'il existe une telle paire dont la distance est inférieure à δ . Observez que, s'il existe une paire de points ayant une distance inférieure à δ , les deux points de la paire doivent être distant de moins de δ unités de la droite l . Donc, comme le montre la figure 33.11(a), ils doivent tous les deux se trouver dans le ruban de largeur 2δ centré sur la droite l . Pour trouver une telle paire, si elle existe, l'algorithme s'y prend de la manière suivante.

- 1) Il crée un tableau Y' , qui est le tableau Y privé de tous les points qui ne se trouvent pas dans le ruban vertical de largeur 2δ . Le tableau Y' est trié selon les ordonnées, tout comme Y .
- 2) Pour chaque point p du tableau Y' , l'algorithme essaie de trouver des points de Y' se trouvant à une distance de p inférieure à δ . Comme nous le verrons bientôt, il suffit de s'intéresser aux 7 points de Y' qui suivent p . L'algorithme calcule la distance de p à chacun de ces 7 points et mémorise la distance δ' entre les deux points les plus rapprochés parmi ceux de Y' .
- 3) Si $\delta' < \delta$, alors le ruban vertical contient effectivement une paire de points plus rapprochés que celle trouvée par les appels récursifs. Cette paire, ainsi que la distance δ' , sont retournés. Sinon, les deux points les plus rapprochés et leur distance δ trouvés par les appels récursifs sont retournés.

La description ci-dessus néglige quelques détails d'implémentation qui sont nécessaires pour atteindre le temps d'exécution $O(n \lg n)$. Après avoir démontré la validité de l'algorithme, nous montrerons comment implémenter l'algorithme pour atteindre la borne désirée.

b) Validité

La validité de cet algorithme est évidente, hormis pour deux aspects. Primo, en fixant la limite de la récursivité à $|P| \leq 3$, on est certain que l'on n'essaiera jamais de résoudre un sous-problème composé d'un seul point. Le second aspect est qu'il suffit de tester les 7 points qui suivent chaque point p dans le tableau Y' ; nous allons maintenant démontrer cette propriété.

Supposons qu'à un certain niveau de la récursivité, les deux points les plus rapprochés soient $p_G \in P_G$ et $p_D \in P_D$. La distance δ' entre p_G et p_D est alors strictement inférieure à δ . Le point p_G doit se trouver sur la droite l ou à sa gauche, à une distance inférieure à δ . De même, p_D se trouve sur ou à droite de l et à une distance inférieure à δ . Par ailleurs, p_G et p_D ne sont pas verticalement distants l'un de l'autre de plus de δ unités. Donc, comme le montre la figure 33.11(a), p_G et p_D se trouvent dans un rectangle $\delta \times 2\delta$ centré sur la droite l . (Ce rectangle peut aussi contenir d'autres points).

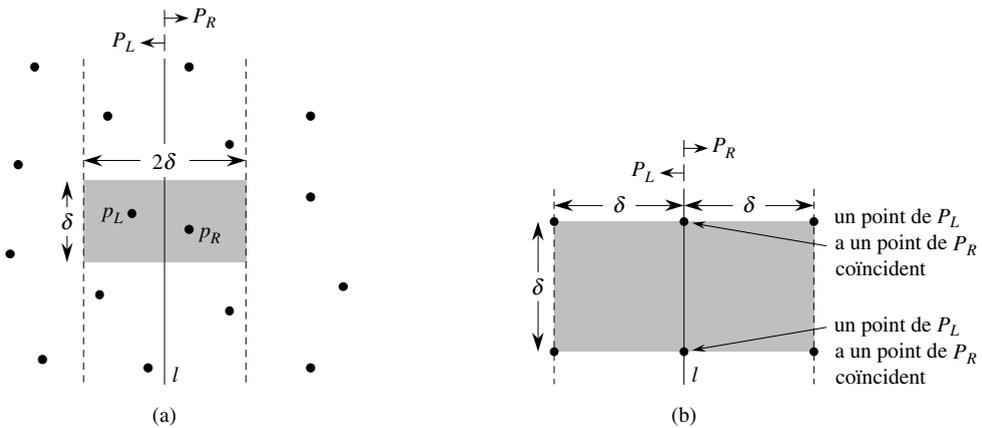


Figure 33.11 Remarques fondamentales pour démontrer que l'algorithme de recherche des deux points les plus rapprochés n'a besoin de tester que les 7 points qui suivent chaque point du tableau Y' . (a) Si $p_G \in P_G$ et $p_D \in P_D$ sont éloignés d'une distance inférieure à δ , ils doivent se trouver à l'intérieur d'un rectangle $\delta \times 2\delta$ centré sur la droite l . (b) Comment 4 points qui sont distants deux à deux d'au moins δ peuvent tous se trouver dans un carré $\delta \times \delta$. Les 4 points de gauche appartiennent à P_G et les 4 points de droite appartiennent à P_D . On peut placer 8 points dans le rectangle $\delta \times 2\delta$ si les points représentés sur la droite l sont en réalité des paires de points coïncidents, l'un appartenant à P_G et l'autre à P_D .

Nous montrerons ensuite qu'au plus 8 points de P peuvent se trouver à l'intérieur de ce rectangle $\delta \times 2\delta$. On considère le carré $\delta \times \delta$ formant la moitié gauche de ce rectangle. Puisque tous les points de P_G sont distants deux à deux d'au moins δ , il y en a au plus 4 qui se trouvent dans ce carré ; la figure 33.11(b) montre comment. De même, il y a au plus 4 points de P_D qui se trouvent à l'intérieur du carré $\delta \times \delta$

formant la moitié droite du rectangle. Donc, il y a au plus 8 points de P qui se trouvent à l'intérieur du rectangle $\delta \times 2\delta$. (Notez que, comme les points de la droite l peuvent être soit dans P_G soit dans P_D , il peut y avoir jusqu'à 4 points sur l . Cette limite est atteinte s'il y a deux paires de points coïncidents telles que chaque paire est constituée d'un point de P_G et d'un point de P_D , une paire est située à l'intersection de l et du côté supérieur du rectangle et l'autre paire est située à l'intersection de l et du côté inférieur du rectangle.)

Maintenant que nous avons montré qu'au plus 8 points de P pouvaient se trouver à l'intérieur du rectangle, il est facile de voir qu'il suffit de tester les 7 points qui suivent chaque point dans le tableau Y' . Supposons toujours que les deux points les plus rapprochés sont p_G et p_D , et supposons aussi sans nuire à la généralité du problème que p_G précède p_D dans le tableau Y' . Alors, même si p_G apparaît le plus tôt possible dans Y' et p_D le plus tard possible, p_D est à l'une des 7 positions qui suivent p_G . Nous avons donc prouvé la validité de l'algorithme de recherche de la paire de points les plus proches.

c) Implémentation et temps d'exécution

Comme nous l'avons remarqué, notre but est d'obtenir la récurrence $T(n) = 2T(n/2) + O(n)$, où $T(n)$ est le temps d'exécution pour un ensemble de n points. La principale difficulté est d'assurer que les tableaux X_G , X_D , Y_G et Y_D , passés aux appels récursifs, sont triés selon la bonne coordonnée, et aussi que le tableau Y' est trié selon les ordonnées. (Notez que, si le tableau X passé à un appel récursif est déjà trié, il est facile d'arriver à un temps linéaire pour la division de l'ensemble P en P_G et P_D .)

L'observation fondamentale est que, dans chaque appel on souhaite former un sous-ensemble trié d'un tableau trié. Par exemple, une invocation particulière reçoit le sous-ensemble P et le tableau Y , trié par ordonnées. Une fois l'ensemble P découpé en P_G et P_D , il faut former les tableaux Y_G et Y_D qui sont triés par ordonnées. Par ailleurs, ces tableaux doivent être construits en temps linéaire. La méthode peut être vue comme l'inverse de la procédure FUSIONNER du tri par fusion de la section 2.3.1 : un tableau trié est maintenant découpé en deux tableaux triés. Le pseudo code suivant en donne le principe.

```

1  longueur[YG] ← longueur[YD] ← 0
2  pour  $i \leftarrow 1$  à longueur[Y]
3      faire si  $Y[i] \in P_G$ 
4          alors longueur[YG] ← longueur[YG] + 1
5               $Y[\text{longueur}[Y_G]] \leftarrow Y[i]$ 
6          sinon longueur[YD] ← longueur[YD] + 1
7               $Y[\text{longueur}[Y_D]] \leftarrow Y[i]$ 

```

On examine tout simplement les points du tableau Y dans l'ordre. Si un point $Y[i]$ se trouve dans P_G , on l'ajoute à la fin du tableau Y_G ; sinon, on l'ajoute à la fin du tableau Y_D . Un pseudo code similaire permet de construire les tableaux X_G , X_D et Y' .

La seule question restante est de savoir comment obtenir les points triés la première fois. Il suffit de les **pré-trier** ; autrement dit, on les trie une fois pour toutes *avant* le premier appel récursif. Ces tableaux triés sont passés au premier appel récursif, et ils sont peu à peu rognés par les appels récursifs suivants. Le pré tri ajoute un supplément de $O(n \lg n)$ au temps d'exécution ; mais ensuite, chaque étape de la récursivité prend un temps linéaire, appels récursifs non compris. Donc, si l'on appelle $T(n)$ le temps d'exécution de chaque étape récursive et $T'(n)$ le temps d'exécution de l'algorithme tout entier, on obtient $T'(n) = T(n) + O(n \lg n)$ et

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{si } n > 3, \\ O(1) & \text{si } n \leq 3. \end{cases}$$

Donc, $T(n) = O(n \lg n)$ et $T'(n) = O(n \lg n)$.

Exercices

33.4.1 Le professeur Houdini a concocté un modèle qui permet à l'algorithme de la paire de points les plus proches de ne tester que les 5 points qui suivent chaque point du tableau Y' . Le principe est de toujours placer les points de la droite l dans l'ensemble P_G . Il ne peut alors pas y avoir de paires de points coïncidents sur la droite l , avec un point dans P_G et un dans P_D . Donc, il y a au plus 6 points qui se trouvent dans le rectangle $\delta \times 2\delta$. Quelle est la faille dans le schéma du professeur ?

33.4.2 Sans augmenter le temps d'exécution asymptotique de l'algorithme, montrer comment garantir que l'ensemble des points passés au tout premier appel récursif ne contient pas de points coïncidents. Démontrer qu'il suffit alors de tester les points situés aux 5 positions qui suivent chaque point dans le tableau Y' .

33.4.3 La distance entre deux points peut être définie autrement qu'avec le modèle euclidien. Dans le plan, la **norme** L_m entre les points p_1 et p_2 est donnée par $((x_1 - x_2)^m + (y_1 - y_2)^m)^{1/m}$. La distance euclidienne est donc une distance L_2 . Modifier l'algorithme de recherche des deux points les plus proches pour qu'il utilise la norme L_1 , également appelée **norme de Manhattan**.

33.4.4 Étant donnés deux points p_1 et p_2 du plan, la norme L_∞ entre eux est donnée par $\max(|x_1 - x_2|, |y_1 - y_2|)$. Modifier l'algorithme de recherche des deux points les plus rapprochés pour qu'il utilise la norme L_∞ .

33.4.5 Trouver comment modifier l'algorithme de la paire de points les plus proches pour qu'il n'ait plus à pré-trier le tableau Y (sans pour autant altérer le temps d'exécution $O(n \lg n)$). (*conseil* : Fusionner les tableaux triés Y_L et Y_R pour former le tableau trié Y .)

PROBLÈMES

33.1. Couches convexes

Étant donné un ensemble Q de points du plan, on définit les *couches convexes* de Q par récurrence. La première couche de Q est constituée des points de Q qui sont des sommets de $EC(Q)$. Pour $i > 1$, on définit Q_i comme étant l'ensemble des points de Q privé de tous les points situés dans les couches convexes $1, 2, \dots, i - 1$. Alors, la i ème couche convexe de Q est $EC(Q_i)$ si $Q_i \neq \emptyset$, et n'est pas définie sinon.

- a. Donner un algorithme en temps $O(n^2)$ pour trouver les couches convexes d'un ensemble de n points.
- b. Démontrer qu'il faut un temps $\Omega(n \lg n)$ pour calculer les couches convexes d'un ensemble de n points avec un modèle de calcul qui demande un temps $\Omega(n \lg n)$ pour trier n nombres réels.

33.2. Couches maximales

Soit Q un ensemble de n points du plan. On dit que le point (x, y) *domine* le point (x', y') si $x \geq x'$ et $y \geq y'$. Un point de Q qui n'est dominé par aucun autre point de Q est dit *maximal*. Notez que Q peut contenir plusieurs points maximaux, qui peuvent être organisés en *couches maximales* de la façon suivante. La première couche maximale L_1 est l'ensemble des points maximaux de Q . Pour $i > 1$, la i ème couche maximale L_i est l'ensemble des points maximaux de $Q - \bigcup_{j=1}^{i-1} L_j$.

On suppose que Q possède k couches maximales non vides et on appelle y_i l'ordonnée du point le plus à gauche de L_i pour $i = 1, 2, \dots, k$. Pour l'instant, supposons que deux points quelconques de Q ne possèdent pas la même abscisse, ni la même ordonnée.

- a. Montrer que $y_1 > y_2 > \dots > y_k$.

On considère un point (x, y) qui est à gauche de chaque point de Q et pour lequel y est distinct de l'ordonnée de chaque point de Q . Soit $Q' = Q \cup \{(x, y)\}$.

- b. Soit j l'indice minimal tel que $y_j < y$, sauf quand $y < y_k$, auquel cas $j = k + 1$. Montrer que les couches maximales de Q' sont définies ainsi :
 - Si $j \leq k$, alors les couches maximales de Q' sont les mêmes que les couches maximales de Q , excepté que L_j inclut aussi (x, y) en tant que nouveau point le plus à gauche.
 - Si $j = k + 1$, alors les k premières couches maximales de Q' sont les mêmes que pour Q , mais en plus, Q' possède une $(k + 1)$ ième couche maximale non vide : $L_{k+1} = \{(x, y)\}$.
- c. Décrire un algorithme à temps $O(n \lg n)$ pour calculer les couches maximales d'un ensemble Q de n points. (*conseil* : Déplacer une droite de balayage de la droite vers la gauche.)

- d. Quelles sont les difficultés qui apparaissent quand on autorise les points d'entrée à avoir les mêmes abscisses ou les mêmes ordonnées ? Suggérer un moyen de résoudre ce type de problèmes.

33.3. Fantômes et chasseurs de fantômes

Une équipe de n chasseurs de fantômes combat n fantômes. Chaque chasseur est armé d'un canon à protons, capable d'éradiquer un fantôme d'un coup de rayon. Un rayon se propage en ligne droite et termine sa course en touchant le fantôme. Les chasseurs de fantômes mettent au point la stratégie suivante. Il provoqueront chaque fantôme en duel, formant ainsi n paires chasseur-fantôme, puis chaque chasseur enverra un rayon en direction du fantôme qu'il s'est attribué. Comme nous le savons tous, il est *très* dangereux de faire se croiser deux rayons et les paires doivent donc être calculés de manière à éviter que des rayons se croisent.

On suppose que la position de chaque chasseur et de chaque fantôme est un point fixe du plan et que trois positions ne sont jamais alignées.

- Montrer qu'il existe une droite passant par un chasseur et par un fantôme telle que le nombre de chasseurs d'un côté de la droite soit égal au nombre de fantômes du même côté. Décrire une manière de trouver cette droite en temps $O(n \lg n)$.
- Donner un algorithme à temps $O(n^2 \lg n)$ pour appairer chasseurs et fantômes de façon qu'aucun rayon ne se croise.

33.4. Ramassage de baguettes

Le professeur Schmilblick dispose d'un ensemble de n baguettes, posées les unes sur les autres selon une certaine configuration. Chaque baguette est définie par ses extrémités, et chaque extrémité est un triplet de coordonnées (x, y, z) . Aucune baguette n'est verticale. Le professeur veut ramasser toutes les baguettes, une à la fois, avec la contrainte qu'il ne peut ramasser une baguette que s'il n'y en pas d'autre posée dessus.

- Donner une procédure qui prend en entrée deux baguettes a et b , puis signale si a est dessus b , dessous b ou ni l'un ni l'autre.
- Décrire un algorithme efficace qui indique si l'on peut ramasser toutes les baguettes, et si oui, qui fournisse une séquence licite de ramassages de baguette.

33.5. Distributions à enveloppes non denses

On considère le problème consistant à calculer l'enveloppe convexe d'un ensemble de points du plan qui ont été placés selon une distribution aléatoire connue. Parfois, le nombre de points (ou taille) de l'enveloppe convexe de n points produits par une telle distribution a une espérance de $O(n^{1-\varepsilon})$, pour une certaine constante $\varepsilon > 0$. On dit que ce type de distribution est à *enveloppes non denses*. Parmi les distributions non denses en enveloppes, on peut citer :

- Les points sont pris uniformément dans un disque de rayon unitaire. L'enveloppe convexe a une taille attendue de $\Theta(n^{1/3})$.
 - Les points sont pris uniformément à l'intérieur d'un polygone convexe à k côtés, pour une constante k quelconque. L'enveloppe convexe a une taille attendue de $\Theta(\lg n)$.
 - Les points sont placés selon une distribution normale à deux dimensions. L'enveloppe convexe a une taille attendue de $\Theta(\sqrt{\lg n})$.
- a. Étant donnés deux polygones convexes à n_1 et n_2 sommets respectivement, montrer comment calculer l'enveloppe convexe des $n_1 + n_2$ points en temps $O(n_1 + n_2)$. (Les polygones peuvent se recouvrir.)
 - b. Montrer que l'enveloppe convexe d'un ensemble de n points dessinés indépendamment selon une loi de distribution à enveloppes non dense peut être calculée avec un temps attendu de $O(n)$. (*Conseil* : Trouver récursivement les enveloppes convexes des $n/2$ premiers points et des $n/2$ derniers points, puis combiner les résultats.)

NOTES

Ce chapitre effleure à peine les algorithmes et techniques de la géométrie algorithmique. Parmi les livres sur la géométrie algorithmique, citons ceux de Preparata et Shamos [247], Edelsbrunner [83], et O'Rourke [235].

Bien que la géométrie ait été étudiée depuis l'antiquité, le développement d'algorithmes permettant de résoudre des problèmes géométriques est relativement nouveau. Preparata et Shamos remarquent que la première notion de la complexité d'un problème fut donnée par E. Lemoine en 1902. Il étudiait les constructions euclidiennes, celles faites à la règle et au compas, et il avait défini un ensemble de cinq primitives : placer une jambe du compas sur un point donné ; placer une jambe du compas sur une droite donnée ; tracer un cercle ; faire passer le bord d'une règle par un point donné ; tracer une droite. Lemoine s'intéressait au nombre de primitives nécessaires pour effectuer une construction donnée ; il appelait cette quantité la « simplicité » de la construction.

L'algorithme de la section 33.2, qui détermine si deux segments sont sécants, est dû à Shamos et Hoey [275].

La version originale du balayage de Graham est donnée par [130]. L'algorithme du paquet cadeau est dû à Jarvis [165]. En se servant d'un arbre de décision comme modèle de calcul, Yao [318] a établi un minorant en $\Omega(n \lg n)$ pour le temps d'exécution d'un algorithme quelconque de calcul d'enveloppe convexe. Lorsque le nombre de sommets h de l'enveloppe convexe est pris en compte, l'algorithme des greffes successives de Kirkpatrick et Seidel [180], qui prend un temps $O(n \lg h)$, est optimal asymptotiquement.

L'algorithme diviser-pour-régner à temps $O(n \lg n)$ permettant de trouver les deux points les plus proches est dû à Shamos, et il apparaît dans Preparata et Shamos [247]. Preparata et Shamos montrent également que l'algorithme est asymptotiquement optimal dans un modèle basé sur les arbres de décision.

Chapitre 34

NP-complétude

Tous les algorithmes étudiés jusqu'ici étaient des *algorithmes à temps polynomial* : sur des entrées de taille n , leur temps d'exécution dans le pire des cas est $O(n^k)$ pour une certaine constante k . Il est naturel de se demander si *tous* les problèmes peuvent être résolus en temps polynomial. La réponse est non. Par exemple, il existe des problèmes, tel le fameux « problème de l'arrêt de Turing d'une machine », qui ne peuvent être résolus par aucun ordinateur, quel que soit le temps qu'il y passe. Il existe aussi des problèmes qui peuvent être résolus, mais pas en temps $O(n^k)$ pour une constante k quelconque. De manière générale, on considère que les problèmes résolubles par des algorithmes à temps polynomial sont traitables (faciles) et que les problèmes nécessitant un temps suprapolynomial sont intraitables (difficiles).

Ce chapitre va présenter une classe de problèmes intéressante, appelés problèmes « NP-complets », dont le statut est inconnu. Aucun algorithme à temps polynomial n'a encore été découvert pour un problème NP-complet, mais personne n'a pu prouver pour l'instant qu'il ne peut pas y avoir d'algorithme à temps polynomial pour un problème NP-complet. Cette question, dite $P \neq NP$, est l'un des problèmes de recherche les plus approfondis et des plus troublants en informatique théorique, depuis qu'il a été posé en 1971.

Un aspect particulièrement frustrant des problèmes NP-complets est que, en surface, plusieurs d'entre eux ressemblent à des problèmes pour lesquels il existe des algorithmes à temps polynomial. Dans chacune des paires suivantes de problèmes, l'un est résoluble en temps polynomial et l'autre est NP-complet, alors même que la différence entre les deux semble être mince :

plus courts chemins et plus longs chemins élémentaires : Au chapitre 24, nous avons vu que, même avec des poids négatifs, nous pouvions trouver des *plus courts* chemins à origine unique dans un graphe orienté $G = (S, A)$ avec un temps $O(SA)$. En revanche, trouver le chemin élémentaire le *plus long* entre deux sommets est NP-complet. En fait, il est NP-complet même si tous les poids des arcs sont 1.

tournée eulérienne et circuit hamiltonien : Une *tournée eulérienne* d'un graphe orienté connexe $G = (S, A)$ est un cycle qui traverse chaque *arc* de G exactement une fois, bien qu'il puisse visiter un sommet plus d'une fois. D'après le problème 25.3, on pourra déterminer en seulement $O(A)$ si un graphe a une tournée eulérienne et, en fait, on peut trouver les arcs de la tournée eulérienne en un temps $O(A)$. Un *circuit hamiltonien* d'un graphe orienté $G = (S, A)$ est un circuit qui contient chaque *sommet* de S . Déterminer si un graphe orienté a un circuit hamiltonien, voilà un exemple de problème NP-complet. (Plus loin dans ce chapitre, nous prouverons que le fait de déterminer si un graphe *non orienté* a un cycle hamiltonien est NP-complet.)

satisfaisabilité 2-CNF et satisfaisabilité 3-CNF : Une formule booléenne contient des variables dont les valeurs sont 0 ou 1, des connecteurs booléens tels que \wedge (ET), \vee (OU) et \neg (NON), ainsi que des parenthèses. Une formule booléenne est *satisfaisable* s'il y a une certaine assignation des valeurs 0 et 1 à ses variables qui fait que la formule prend la valeur 1. Nous donnerons une définition plus précise ultérieurement mais, de manière informelle, une formule booléenne est en *forme normale conjonctive d'ordre k*, ou k -CNF, si c'est le ET de clauses de OU de k variables exactement (ou de leurs négations). Ainsi, la formule booléenne $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ est en 2-CNF. (Elle possède l'assignation satisfaisante $x_1 = 1, x_2 = 0, x_3 = 1$.) Il y a un algorithme à temps polynomial pour déterminer si une formule 2-CNF est satisfaisable mais, comme nous le verrons plus loin, le fait de déterminer si une formule 3-CNF est satisfaisable est NP-complet.

a) NP-complétude et les classes P et NP

Tout au long de ce chapitre, nous nous référerons à trois classes de problèmes : P, NP et NPC, la dernière étant la classe des problèmes NP-complets. Nous les présenterons ici de manière informelle, gardant les définitions précises pour plus tard.

La classe P se compose des problèmes qui sont résolubles en temps polynomial. Plus précisément, il existe des problèmes qui peuvent être résolus en temps $O(n^k)$ pour une certaine constante k (n est la taille de l'entrée). La plupart des problèmes vus dans les chapitres précédents appartiennent à P.

La classe NP se compose des problèmes qui sont « vérifiables » en temps polynomial. Cela signifie : si l'on nous donne, d'une façon ou d'une autre, une solution « certifiée », nous pouvons vérifier que cette solution est correcte en un temps qui

est polynomial par rapport à la taille de l'entrée. Par exemple, dans le problème du circuit hamiltonien, étant donné un graphe orienté $G = (S, A)$, une solution certifiée serait une suite $\langle v_1, v_2, v_3, \dots, v_{|S|} \rangle$ de $|S|$ sommets. Il est facile de vérifier en temps polynomial que $(v_i, v_{i+1}) \in A$ pour $i = 1, 2, 3, \dots, |S| - 1$ et que $(v_{|S|}, v_1) \in A$ aussi. Autre exemple, concernant la satisfaisabilité 3-CNF : une solution certifiée serait une assignation de valeurs aux variables. On peut facilement vérifier en temps polynomial si cette assignation satisfait à la formule booléenne.

Tout problème P appartient aussi à NP ; en effet, si un problème appartient à P , alors on peut le résoudre en temps polynomial même si l'on ne nous donne pas de solution. Nous formaliserons cette notion plus loin mais, pour l'instant, contentons-nous de croire que $P \subseteq NP$. La question qui reste ouverte est de savoir si P est ou non un sous-ensemble propre de NP .

De manière informelle, un problème ressort de la classe NPC (nous dirons plus simplement qu'il est *NP-complet*) s'il appartient à NP et s'il est aussi « difficile » que n'importe quel problème de NP . Nous définirons plus loin ce que l'on entend par « être aussi difficile que n'importe quel problème de NP ». Pour l'heure, nous énoncerons sans preuve que s'il existe *un quelconque* problème NP -complet qui est résoluble en temps polynomial, alors *tout* problème NP -complet a un algorithme à temps polynomial.

La plupart des théoriciens de l'informatique pensent que les problèmes NP -complets sont intraitables, pour la raison suivante : compte tenu de la gamme étendue des problèmes NP -complets qui ont été étudiés à ce jour (sans que quiconque ait découvert d'algorithme à temps polynomial pour l'un de ces problèmes), il serait vraiment étonnant que tous ces problèmes soient résolubles en temps polynomial. Et pourtant, compte tenu de tous les efforts qui ont été faits pour démontrer que les problèmes NP -complets sont intraitables (sans que ces efforts aient été couronnés de succès), on ne peut pas exclure la possibilité que les problèmes NP -complets soient résolubles en temps polynomial.

Pour devenir un bon concepteur d'algorithmes, il faut comprendre les rudiments de la théorie de la NP -complétude. Si l'on peut établir qu'un problème est NP -complet, on peut être quasi certain qu'il est intraitable. En tant qu'ingénieur, vous perdrez alors moins de temps en choisissant plutôt de développer un algorithme approché (voir chapitre 35). Par ailleurs, de nombreux problèmes naturels et intéressants qui ne semblent a priori pas plus difficiles que le tri, l'exploration d'un graphe ou le flot dans un réseau de transport sont en réalité NP -complets. Il est donc important d'être familiarisé avec cette classe de problèmes remarquable.

b) Aperçu de la technique par laquelle on montre que des problèmes sont NP-complets

Les techniques employées pour montrer qu'un problème est NP -complet diffèrent des techniques utilisées dans la majeure partie de ce livre pour concevoir et analyser

les algorithmes. Cette différence tient à une raison fondamentale : dans la démonstration qu'un problème est NP-complet, on énonce son degré de difficulté (du moins, tel qu'on l'estime), pas son degré de facilité. On n'essaie pas de prouver l'existence d'un algorithme efficace, mais plutôt la probabilité de la non-existence d'un algorithme efficace. Ainsi, les démonstrations de NP-complétude ressemblent un peu à la démonstration, vue à la section 8.1, concernant le minorant temporel $\Omega(n \lg n)$ de tout algorithme de tri par comparaisons ; les techniques spécifiques utilisées pour démontrer la NP-complétude diffèrent, cependant, de la méthode de l'arbre de décision employée à la section 8.1.

Trois concepts clé nous serviront à prouver qu'un problème est NP-complet :

► Problèmes de décision et problèmes d'optimisation

Nombre de problèmes intéressants sont des *problèmes d'optimisation*, dans lesquels chaque solution réalisable (c'est à dire, « légale ») a une valeur associée et où l'on veut trouver la solution réalisable qui a la valeur optimale. Ainsi, dans un problème que nous appelons PLUS-COURT-CHEMIN, on a un graphe non orienté G et des sommets u et v , et l'on veut trouver le chemin de u à v qui utilise le moins d'arêtes. (En d'autres termes, PLUS-COURT-CHEMIN est le problème du plus court chemin à origine unique dans un graphe non orienté non pondéré.) La NP-complétude ne s'applique pas directement, toutefois, aux problèmes d'optimisation, mais aux *problèmes de décision* dans lesquels la réponse est tout simplement « oui » ou « non » (ou, plus formellement, « 1 » ou « 0 »).

Bien que la démonstration de la NP-complétude d'un problème nous confine dans le domaine des problèmes de décision, il existe une relation commode entre problèmes d'optimisation et problèmes de décision. On peut généralement convertir un problème d'optimisation en un problème de décision, en imposant une borne à la valeur à optimiser. Pour PLUS-COURT-CHEMIN, par exemple, un problème de décision associé, que nous appellerons CHEMIN, est le suivant : étant donné un graphe orienté G , des sommets u et v et un entier k , il existe un chemin de u à v composé d'au plus k arcs.

La relation entre un problème d'optimisation et le problème de décision associé travaille en notre faveur quand nous essayons de montrer que le problème d'optimisation est « difficile ». La raison en est que le problème de décision est, en un sens, « plus facile », ou au moins « pas plus difficile ». À titre d'exemple spécifique, on peut résoudre CHEMIN en résolvant PLUS-COURT-CHEMIN puis en comparant le nombre d'arcs du plus court chemin trouvé et la valeur du paramètre k du problème de décision. Autrement dit, si un problème d'optimisation est facile, le problème de décision associé est facile lui aussi. Énoncé d'une façon plus en rapport avec la NP-complétude : si l'on fournit la preuve qu'un problème de décision est difficile, on fournit de ce fait la preuve que le problème d'optimisation associé est difficile. Donc, même si elle ne s'intéresse qu'aux problèmes de décision, la théorie de la NP-complétude a souvent des répercussions sur les problèmes d'optimisation.

► Réductions

Cette notion de montrer qu'un problème n'est pas plus difficile ou plus facile qu'un autre s'applique même quand les deux problèmes sont des problèmes de décision. Nous exploiterons cette idée dans la quasi totalité des démonstrations de NP-complétude, et ce de la façon suivante. Soit un problème de décision, disons A , que l'on voudrait résoudre en temps polynomial. L'entrée d'un problème particulier est dite *instance* de ce problème ; ainsi, dans CHEMIN, une instance serait un graphe particulier G , des sommets particuliers u et v de G , et un entier particulier k . Supposons maintenant qu'il y ait un autre problème de décision, disons B , que nous savons déjà comment résoudre en temps polynomial. Enfin, supposons que nous ayons une procédure qui transforme toute instance α de A en une certaine instance β de B et qui a les caractéristiques suivantes :

- 1) La transformation prend un temps polynomial.
- 2) Les réponses sont les mêmes. C'est à dire, la réponse pour α est « oui » si et seulement si la réponse pour β est « oui ».

Une telle procédure porte le nom *de réduction* à temps polynomial et, comme le montre la figure 34.1, elle donne un moyen de résoudre le problème A en temps polynomial :

- 1) Étant donnée une instance α de A , utiliser une réduction à temps polynomial pour la transformer en une instance β de B .
- 2) Exécuter l'algorithme de décision à temps polynomial de B sur l'instance β .
- 3) Utiliser la réponse pour β comme réponse pour α .

Si chacune de ces étapes prend un temps polynomial, il en est de même de l'ensemble ; on a donc un moyen de décider pour α en temps polynomial. Autrement dit, en « réduisant » la résolution du problème A à celle du problème B , on se sert de la « facilité » de B pour prouver la « facilité » de A .

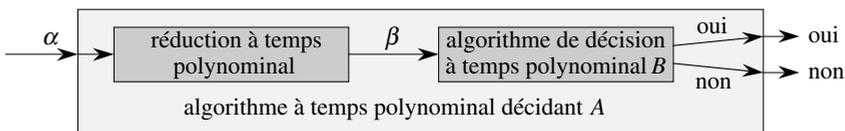


Figure 34.1 Utilisation d'une réduction à temps polynomial pour résoudre un problème de décision A en temps polynomial à partir d'un algorithme de décision à temps polynomial associé à un autre problème B . En temps polynomial, on transforme une instance α de A en une instance β de B , on résout B en temps polynomial, puis on utilise la réponse pour β comme réponse pour α .

En nous rappelant que la NP-complétude consiste à démontrer le niveau de difficulté d'un problème et non son niveau de facilité, nous pouvons utiliser des réductions

à temps polynomial, en sens inverse, pour prouver qu'un problème est NP-complet. Poussons l'idée un cran plus loin et montrons comment nous pourrions employer des réductions à temps polynomial pour prouver qu'il ne peut exister d'algorithme à temps polynomial pour un certain problème B . Supposons que l'on ait un problème de décision A pour lequel on sait déjà qu'il ne peut pas exister d'algorithme à temps polynomial. (Nous laisserons de côté, pour l'instant, le problème de savoir comment trouver un tel problème A .) Supposons, en outre, que l'on ait une réduction à temps polynomial qui transforme des instances de A en instances de B . On peut alors utiliser un raisonnement simple par l'absurde pour prouver qu'il ne peut pas exister d'algorithme à temps polynomial pour B . Supposons le contraire, c'est à dire que B ait un algorithme à temps polynomial. Alors, en utilisant la méthode illustrée à la figure 34.1, on aurait un moyen de résoudre A en temps polynomial, ce qui est contraire à l'hypothèse selon laquelle il n'y a pas d'algorithme à temps polynomial pour A .

Pour la NP-complétude, on ne peut pas partir du principe qu'il n'y a absolument pas d'algorithme à temps polynomial pour le problème A . Cependant, la méthodologie de la démonstration est la même : nous prouvons que le problème B est NP-complet en supposant que le problème A est lui aussi NP-complet.

► Un premier problème NP-complet

Comme la technique de réduction suppose que l'on utilise un problème garanti NP-complet pour prouver qu'un autre problème est NP-complet, il nous faut un « premier » problème NP-complet. Le problème que nous utiliserons est celui de la satisfaisabilité de circuit : on se donne un circuit combinatoire booléen, composé de portes ET, OU et NON, et l'on veut savoir s'il existe un ensemble d'entrées booléennes qui fait que ce circuit donne en sortie la valeur 1. Nous montrerons que ce premier problème est NP-complet à la section 34.3.

c) Résumé du chapitre

Ce chapitre étudie les aspects de la NP-complétude qui sont le plus en rapport avec l'analyse des algorithmes. A la section 34.1, on formalise la notion de « problème » et l'on définit la classe de complexité P des problèmes de décision résolubles en temps polynomial. On voit également comment ces notions s'intègrent dans le cadre de la théorie des langages formels. La section 34.2 définit la classe NP des problèmes de décision dont les solutions peuvent être vérifiées en temps polynomial. Elle pose également la question $P \neq NP$ de façon formelle.

La section 34.3 montre comment étudier les relations entre problèmes via des « réductions » en temps polynomial. Elle définit la NP-complétude et ébauche une démonstration que le problème de la « satisfaisabilité de circuit » est NP-complet. Ayant trouvé un problème NP-complet, nous montrerons à la section 34.4 comment prouver que d'autres problèmes sont NP-complets beaucoup plus facilement par la

technique des réductions. Cette méthodologie est illustrée en montrant que deux problèmes de satisfaisabilité de formule sont NP-complets. La section 34.5 montrera la NP-complétude de plusieurs autres problèmes.

34.1 TEMPS POLYNOMIAL

Nous commençons notre étude de la NP-complétude en formalisant la notion de problèmes résolubles en temps polynomial. Ces problèmes sont en général considérés comme traitables, mais pour des raisons philosophiques et non mathématiques. On peut présenter trois arguments en ce sens.

Primo, bien qu'il soit raisonnable de considérer comme intraitable un problème qui demande un temps $\Theta(n^{100})$, il existe en pratique très peu de problèmes qui requièrent un temps d'un ordre de grandeur polynomiale aussi élevé. Les problèmes calculables en temps polynomial rencontrés en pratique demandent beaucoup moins de temps. L'expérience montre que, une fois que l'on a découvert un algorithme à temps polynomial pour un problème, généralement on trouve ensuite des algorithmes plus performants. Même si le meilleur algorithme du moment tourne en temps $\Theta(n^{100})$, il y a de fortes chances pour que l'on découvre incessamment sous peu un algorithme dont le temps d'exécution est bien meilleur.

Secundo, pour de nombreux modèles de calcul raisonnables, un problème qui peut être résolu en temps polynomial dans un modèle peut également l'être en temps polynomial dans un autre. Par exemple, la classe des problèmes résolubles en temps polynomial par la machine séquentielle à accès direct utilisée tout au long de ce livre est identique à la classe des problèmes résolubles en temps polynomial sur les machines abstraites de Turing ⁽¹⁾. C'est aussi la même classe que celle des problèmes résolubles en temps polynomial sur un ordinateur parallèle quand le nombre de processeurs croît de façon polynomiale avec la taille de l'entrée.

Tertio, la classe des problèmes résolubles en temps polynomial possède de sympathiques propriétés de fermeture, puisque les polynômes sont fermés pour l'addition, la multiplication et la composition. Par exemple, si le résultat d'un algorithme à temps polynomial est redirigé vers l'entrée d'un autre, l'algorithme composé est encore polynomial. Si un autre algorithme à temps polynomial effectue un nombre constant d'appels à des sous-programmes à temps polynomial, le temps d'exécution de l'algorithme composé est polynomial.

d) Problèmes abstraits

Pour comprendre la classe des problèmes résolubles en temps polynomial, il faut d'abord avoir une notion formelle de ce qu'est un « problème ». On définit un **problème abstrait** Q comme étant une relation binaire sur un ensemble I d'**instances** de

(1) Voir Hopcroft et Ullman [156] ou Lewis et Papadimitriou [204] pour un traitement complet du modèle de la machine de Turing.

problème et un ensemble S de **solutions** de problème. Par exemple, une instance de PLUS-COURT-CHEMIN est un triplet composé d'un graphe et de deux sommets. Une solution est une séquence de sommets du graphe ; si cette séquence est vide, c'est qu'il n'existe aucun chemin dans le graphe entre ces deux sommets. Le problème PLUS-COURT-CHEMIN lui-même est la relation qui associe chaque instance d'un graphe et deux sommets à un plus court chemin reliant les deux sommets dans le graphe. Comme les plus courts chemins ne sont pas nécessairement uniques, une instance de problème donnée peut avoir plus d'une solution.

Cette formulation d'un problème abstrait est trop générale pour notre propos. Comme précédemment mentionné, la théorie de la NP-complétude se restreint aux **problèmes de décision** : ceux qui ont une solution oui/non. Dans ce cas, on peut voir un problème de décision abstrait comme une fonction qui fait correspondre l'ensemble des instances I à l'ensemble des solutions $\{0, 1\}$. Par exemple, un problème de décision lié à PLUS-COURT-CHEMIN est le problème CHEMIN que nous avons vu précédemment. Si $i = \langle G, u, v, k \rangle$ est une instance du problème de décision CHEMIN, alors $\text{CHEMIN}(i) = 1$ (oui) si un plus court chemin de u à v a au plus k arcs, et $\text{CHEMIN}(i) = 0$ (non) sinon. De nombreux problèmes abstraits ne sont pas des problèmes de décision, mais plutôt des **problèmes d'optimisation**, dans lesquels une certaine valeur doit être minimisée ou maximisée. Comme nous l'avons vu, c'est généralement une tâche simple que de transformer un problème d'optimisation en un problème de décision qui n'est pas plus difficile

e) Encodages

Si un programme informatique doit résoudre un problème abstrait, les instances du problème doivent être représentées sous une forme compréhensible par le programme. Un **encodage** d'un ensemble S d'objets abstraits est une application e de S vers l'ensemble des chaînes binaires ⁽²⁾. Par exemple, l'ensemble $\mathbf{N} = \{0, 1, 2, 3, 4, \dots\}$ des entiers naturels représentés sous forme de chaînes $\{0, 1, 10, 11, 100, \dots\}$ nous est familier. Avec cet encodage, $e(17) = 10001$. N'importe qui s'étant déjà intéressé aux représentations des caractères du clavier dans un ordinateur sait ce que sont les codes ASCII ou EBCDIC. Dans le code ASCII, $e(A) = 1000001$. Même un objet composé peut être codé comme une chaîne binaire en combinant les représentations des parties qui le constituent. Les polygones, les graphes, les fonctions, les couples, les programmes, tout cela peut être encodé sous la forme de chaînes binaires.

Donc, un algorithme informatique qui « résout » un problème de décision abstrait prend en fait en entrée un encodage d'une instance du problème. Un problème dont l'ensemble des instances est l'ensemble des chaînes binaires est appelé **problème concret**. On dit qu'un algorithme **résout** un problème concret en temps $O(T(n))$ si, sur une instance i du problème de longueur $n = |i|$, l'algorithme est capable de produire

(2) Il n'est pas nécessaire que le codomaine de e soit l'ensemble des chaînes binaires ; un ensemble de chaînes sur un alphabet fini comportant au moins 2 symboles fera l'affaire.

la solution en un temps $O(T(n))$.⁽³⁾ Un problème concret est donc **résoluble en temps polynomial**, s'il existe un algorithme permettant de le résoudre en temps $O(n^k)$ pour une certaine constante k .

Nous pouvons à présent définir formellement la **classe de complexité P** comme l'ensemble des problèmes de décision concrets qui sont résolubles en temps polynomial.

On peut utiliser l'encodage pour passer des problèmes abstraits aux problèmes concrets. Étant donné un problème de décision abstrait Q reliant un ensemble d'instances I à $\{0, 1\}$, un encodage $e : I \rightarrow \{0, 1\}^*$ permet d'induire un problème de décision concret associé, que nous noterons $e(Q)$.⁽⁴⁾ Si la solution à une instance de problème abstrait $i \in I$ est $Q(i) \in \{0, 1\}$, alors la solution de l'instance du problème concret $e(i) \in \{0, 1\}^*$ est aussi $Q(i)$. Techniquement parlant, il peut exister certaines chaînes binaires qui ne représentent aucune instance cohérente de problème abstrait. Par commodité, on supposera que toute chaîne de ce type a pour image 0. Donc, le problème concret produit les mêmes solutions que le problème abstrait sur les instances de chaîne binaire qui représentent les encodages des instances de problème abstrait.

On souhaiterait étendre la définition de la résolubilité en temps polynomial des problèmes concrets aux problèmes abstraits en utilisant les encodages comme passerelles, mais on voudrait que la définition soit indépendante de tout encodage particulier. Autrement dit, l'efficacité de la résolution d'un problème ne doit pas dépendre de la façon dont le problème est encodé. Malheureusement, elle en est très dépendante. Par exemple, supposons qu'un entier k doit être fourni comme entrée unique d'un algorithme, et que le temps d'exécution de l'algorithme soit $\Theta(k)$. Si l'entier k est fourni en **unaire** (chaîne de k 1), le temps d'exécution de l'algorithme est $O(n)$ sur des entrées de longueur n , c'est-à-dire polynomial. En revanche, si l'on utilise la représentation binaire plus naturelle de l'entier k , la longueur de l'entrée est $n = \lfloor \lg k \rfloor + 1$. Dans ce cas, le temps d'exécution de l'algorithme est $\Theta(k) = \Theta(2^n)$, qui est exponentiel par rapport à la taille de l'entrée. Donc, selon l'encodage choisi, l'algorithme peut s'exécuter en temps polynomial ou suprapolynomial.

L'encodage d'un problème abstrait est donc très important pour notre compréhension du temps polynomial. On ne peut pas vraiment parler de résolution d'un problème abstrait sans commencer par spécifier un encodage. Néanmoins, dans la pratique, si l'on élimine les encodages « coûteux » comme les encodages unaires, l'encodage d'un problème intervient peu sur la question de savoir si la résolution peut s'effectuer en temps polynomial. Par exemple, la représentation des entiers en base 3 au lieu de la base 2 n'a aucun effet sur le caractère polynomial du temps de

(3) On suppose que le résultat de l'algorithme est séparé de l'entrée. Comme la production de chaque bit du résultat demande au moins une unité de temps et qu'il existe $O(T(n))$ étapes temporelles, la taille du résultat est $O(T(n))$.

(4) Comme nous le verrons bientôt, $\{0, 1\}^*$ représente l'ensemble de toutes les chaînes composées de symbole pris dans l'ensemble $\{0, 1\}$.

résolution d'un problème, puisqu'un entier représenté en base 3 peut être converti en base 2 en temps polynomial.

On dit qu'une fonction $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ est **calculable en temps polynomial** s'il existe un algorithme à temps polynomial A qui, étant donnée une entrée $x \in \{0, 1\}^*$ quelconque, produit le résultat $f(x)$. Pour un certain ensemble I d'instances de problème, on dit que deux encodages e_1 et e_2 sont **reliés polynomialement** s'il existe deux fonctions calculables en temps polynomial f_{12} et f_{21} telles que, pour tout $i \in I$, on a $f_{12}(e_1(i)) = e_2(i)$ et $f_{21}(e_2(i)) = e_1(i)$.⁽⁵⁾ Autrement dit, l'encodage $e_2(i)$ peut être calculé à partir de l'encodage $e_1(i)$ par un algorithme à temps polynomial, et vice versa. Si deux encodages e_1 et e_2 d'un problème abstrait sont reliés polynomialement, l'utilisation de l'un ou l'autre ne modifie pas le caractère polynomial ou non du temps de résolution du problème, comme le montre le lemme suivant.

Lemme 34.1 *Soit Q un problème de décision abstrait sur un ensemble d'instances I , et soient e_1 et e_2 des encodages reliés polynomialement sur I . Alors, $e_1(Q) \in P$ si et seulement si $e_2(Q) \in P$.*

Démonstration : Il suffit de démontrer une seule direction de l'équivalence, puisque la direction inverse est symétrique. Supposons donc que $e_1(Q)$ puisse être résolu en temps $O(n^k)$ pour une certaine constante k . Supposons, de plus, que, pour toute instance i du problème, l'encodage $e_1(i)$ puisse être calculé à partir de l'encodage $e_2(i)$ en temps $O(n^c)$ pour une certaine constante c , où $n = |e_2(i)|$. Pour résoudre le problème $e_2(Q)$, sur l'entrée $e_2(i)$, on commence par calculer $e_1(i)$ puis on exécute l'algorithme pour $e_1(Q)$ sur $e_1(i)$. Combien de temps cela prend-il ? La conversion des encodages prend un temps $O(n^c)$, et donc $|e_1(i)| = O(n^c)$, puisque le résultat d'un ordinateur séquentiel ne peut pas être plus long que son temps d'exécution. La résolution du problème sur $e_1(i)$ prend un temps $O(|e_1(i)|^k) = O(n^{ck})$, ce qui est polynomial puisque c et k sont des constantes. \square

Donc, le fait que les instances d'un problème abstrait soient encodées en binaire ou en base 3 n'affecte pas sa « complexité », c'est-à-dire sa capacité à être ou non résolu en temps polynomial ; mais si les instances sont encodées en unaire, sa complexité peut changer. Pour pouvoir effectuer les conversions indépendamment de l'encodage, on suppose généralement que les instances du problème sont encodées d'une façon raisonnable et concise, sauf spécification contraire. Plus précisément, nous supposons que l'encodage d'un entier peut être relié polynomialement à sa représentation binaire, et que l'encodage d'un ensemble fini est relié polynomialement à son encodage sous la forme d'une liste de ses éléments, encadrés par des accolades et séparés par des virgules. (ASCII est un modèle d'encodage de ce type.) Muni d'un tel encodage « standard », on peut déduire des encodages raisonnables pour d'autres objets

(5) Techniquement parlant, on exige aussi que les fonctions f_{12} et f_{21} « fassent correspondre des non instances à des non instances ». Une **non instance** d'un encodage e est une chaîne $x \in \{0, 1\}^*$ telle qu'il n'existe pas d'instance i pour laquelle $e(i) = x$. On exige que $f_{12}(x) = y$ pour toute non instance x d'encodage e_1 , où y est une certaine non instance de e_2 , et que $f_{21}(x') = y'$ pour toute non instance x' de e_2 , où y' est une certaine non instance de e_1 .

mathématiques comme les n -uplets, les graphes et les formules. Pour indiquer l'encodage standard d'un objet, nous l'entourerons l'objet par des accolades angulaires. Donc $\langle G \rangle$ signale un encodage standard pour le graphe G .

Tant que l'on fait appel implicitement à un encodage relié polynomialement à cet encodage standard, on peut parler directement de problèmes abstraits, sans faire référence à un encodage particulier, puisqu'on sait que le choix de l'encodage n'a aucun effet sur le caractère polynomial ou non du temps de résolution d'un problème abstrait. Désormais, on supposera le plus souvent que toutes les instances d'un problème sont des chaînes binaires encodées à l'aide de l'encodage standard, sauf spécification contraire. On négligera aussi le plus souvent la distinction entre problèmes abstraits et problèmes concrets. Cela dit, le lecteur devra faire attention aux problèmes qui surviennent en pratique, dans lesquels l'encodage standard n'est pas évident et où la façon d'encoder a son importance.

f) Un cadre pour les langages formels

L'un des intérêts pratiques de s'en tenir aux problèmes de décision est qu'ils facilitent l'emploi des mécanismes de la théorie des langages formels. Il n'est pas inutile ici de revoir quelques définitions de cette théorie. Un **alphabet** Σ est un ensemble fini de symboles. Un **langage** L sur Σ est un ensemble quelconque de chaînes construites à partir de symboles de Σ . Par exemple, si $\Sigma = \{0, 1\}$, l'ensemble $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$ est le langage des représentations binaires des nombres premiers. La **chaîne vide** est notée ε , et le **langage vide** est noté \emptyset . Le langage de toutes les chaînes sur Σ est noté Σ^* . Par exemple, si $\Sigma = \{0, 1\}$, alors $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$ est l'ensemble de toutes les chaînes binaires. Tout langage L sur Σ est un sous-ensemble de Σ^* .

Il existe de nombreuses opérations sur les langages. Les opérations ensemblistes, comme **l'union** et **l'intersection**, se déduisent directement des définitions de la théorie des ensembles. On définit le **complément** de L par $\bar{L} = \Sigma^* - L$. La **concaténation** de deux langages L_1 et L_2 est le langage

$$L = \{x_1x_2 : x_1 \in L_1 \text{ et } x_2 \in L_2\} .$$

La **fermeture** ou **Kleene étoile** d'un langage L est le langage

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots ,$$

où L^k est le langage obtenu par k concaténations successives de L avec lui-même.

Du point de vue de la théorie des langages, l'ensemble des instances d'un problème de décision Q est tout simplement l'ensemble Σ^* , où $\Sigma = \{0, 1\}$. Puisque Q est entièrement caractérisé par les instances du problème qui produisent une réponse 1 (oui), on peut voir Q comme un langage L sur $\Sigma = \{0, 1\}$, où

$$L = \{x \in \Sigma^* : Q(x) = 1\} .$$

Par exemple, le problème de décision CHEMIN a pour langage correspondant

$$\text{CHEMIN} = \{ \langle G, u, v, k \rangle : G = (S, A) \text{ est un graphe non orienté,} \\ u, v \in S, \\ k \geq 0 \text{ est un entier, et} \\ \text{il existe une chaîne de } u \text{ vers } v \text{ dans } G \\ \text{composé d'au plus } k \text{ arêtes} \} .$$

(Par commodité, on utilise parfois le même nom, ici CHEMIN, pour indiquer à la fois un problème de décision et le langage correspondant.)

La théorie des langages formels permet d'exprimer de façon concise la relation entre les problèmes de décision et les algorithmes qui les résolvent. On dit qu'un algorithme *A* **accepte** une chaîne $x \in \{0, 1\}^*$ si, étant donnée une entrée x , l'algorithme produit $A(x) = 1$. Le langage **accepté** par un algorithme *A* est l'ensemble des chaînes acceptées par l'algorithme, soit $L = \{x \in \{0, 1\}^* : A(x) = 1\}$. Un algorithme *A* **rejette** une chaîne x si $A(x) = 0$.

Même si le langage L est accepté par un algorithme *A*, l'algorithme ne rejettera pas forcément une chaîne $x \notin L$. Par exemple, l'algorithme pourra boucler indéfiniment. Un langage L est **décidé** par un algorithme *A* si toute chaîne binaire de L est acceptée par *A* et si toute chaîne binaire n'appartenant pas à L est rejetée par *A*. Un langage L est **accepté en temps polynomial** par un algorithme *A* s'il est accepté par *A* et si, en outre, il existe une constante k telle que, pour toute chaîne x de longueur n appartenant à L , l'algorithme *A* accepte x en temps $O(n^k)$. Un langage L est **décidé en temps polynomial** par un algorithme *A* s'il existe une constante k telle que, pour toute chaîne $x \in \{0, 1\}^*$ de longueur n , l'algorithme décide correctement si $x \in L$ en temps $O(n^k)$. Donc, pour accepter un langage, un algorithme n'a besoin de s'intéresser qu'aux chaînes de L , alors que pour décider un langage, il doit accepter ou rejeter correctement toutes les chaînes de $\{0, 1\}^*$.

Par exemple, le langage CHEMIN peut être accepté en temps polynomial. Un algorithme acceptant en temps polynomial vérifie que G encode un graphe non orienté, vérifie que u et v sont des sommets de G , utilise une recherche en largeur pour calculer le plus court chemin de u à v dans G , puis compare le nombre d'arêtes du plus court chemin obtenu à k . Si G encode un graphe non orienté et que le chemin de u à v a au plus k arêtes, l'algorithme affiche 1 puis s'arrête. Autrement, l'algorithme tourne indéfiniment. Toutefois, cet algorithme ne décide pas CHEMIN, puisqu'il n'affiche jamais 0 explicitement pour les instances dans lesquelles le plus court chemin a plus de k arêtes. Un algorithme de décision pour CHEMIN doit rejeter explicitement les chaînes qui n'appartiennent pas à CHEMIN. Pour un problème de décision tel que CHEMIN, ce type d'algorithme de décision est facile à concevoir : au lieu de tourner indéfiniment quand il n'existe pas de chemin de u à v ayant au plus k arcs, l'algorithme affiche 0 et s'arrête. Pour d'autres problèmes, comme le problème de l'arrêt d'une machine de Turing, il existe un algorithme d'acceptation mais pas d'algorithme de décision.

On peut définir de manière informelle une *classe de complexité* comme un ensemble de langages, dont l'appartenance est déterminée par une *mesure de la complexité*, tel le temps d'exécution d'un algorithme, qui détermine si une chaîne donnée x appartient au langage L . La véritable définition d'une classe de complexité est un peu plus technique (le lecteur intéressé est renvoyé à l'article original de Hartmanis et Stearns [140]).

En utilisant le cadre de la théorie des langages, on peut proposer une autre définition de la classe de complexité P :

$$P = \{L \subseteq \{0, 1\}^* : \text{il existe un algorithme } A \\ \text{qui décide } L \text{ en temps polynomial}\} .$$

En fait, P est aussi la classe des langages qui peuvent être acceptés en temps polynomial.

Théorème 34.2

$$P = \{L : L \text{ est accepté par un algorithme à temps polynomial}\} .$$

Démonstration : Puisque la classe des langages décidés par des algorithmes à temps polynomial est un sous-ensemble de la classe des langages acceptés par des algorithmes à temps polynomial, il suffit de montrer que, si L est accepté par un algorithme à temps polynomial, il est décidé par un algorithme à temps polynomial. Soit L le langage accepté par un certain algorithme A à temps polynomial. On utilise une démonstration classique par « simulation » pour construire un autre algorithme à temps polynomial A' qui décide L . Comme A accepte L en temps $O(n^k)$, pour une certaine constante k , il existe également une constante c telle que A accepte L en au plus $T = cn^k$ étapes. Pour une chaîne d'entrée x quelconque, l'algorithme A' simule l'action de A pendant le temps T . A la fin du délai T , l'algorithme A' observe le comportement de A . Si A a accepté x , alors A' accepte x en affichant un 1. Si A n'a pas accepté x , alors A' rejette x en affichant un 0. Le travail supplémentaire dû à la simulation de A par A' n'augmente pas le temps d'exécution de plus d'un facteur polynomial, et donc A' est un algorithme à temps polynomial qui décide L . \square

Notez que la démonstration du théorème 34.2 est non constructive. Pour un langage $L \in P$ donné, on peut ne pas connaître de borne sur le temps d'exécution de l'algorithme A qui accepte L . Néanmoins, on sait qu'une telle borne existe, et donc qu'il existe un algorithme A' capable de vérifier la borne, même si l'on n'est pas en mesure de trouver facilement l'algorithme A' .

Exercices

34.1.1 Définir le problème d'optimisation LONGUEUR-PLUS-LONG-CHEMIN sous la forme d'une relation qui associe chaque instance d'un graphe non orienté et de deux sommets au nombre d'arêtes du plus long chemin élémentaire entre les deux sommets. Définir le problème de décision PLUS-LONG-CHEMIN = $\{\langle G, u, v, k \rangle : G = (S, A) \text{ est un graphe non orienté, } u, v \in S, k \geq 0 \text{ est un entier, et il existe un chemin élémentaire de}$

u vers v dans G composé d'au moins k arêtes}. Montrer que le problème d'optimisation LONGUEUR-PLUS-LONG-CHEMIN peut être résolu en temps polynomial si et seulement si PLUS-LONG-CHEMIN \in P.

34.1.2 Donner une définition formelle du problème consistant à trouver le plus long cycle élémentaire dans un graphe non orienté. Donner un problème de décision associé. Donner le langage correspondant au problème de décision.

34.1.3 Donner un encodage formel des graphes orientés, sous la forme de chaînes binaires utilisant une représentation par matrice d'adjacence. Même question avec la représentation par listes d'adjacence. Montrer que les deux représentations sont reliées polynomialement.

34.1.4 L'algorithme de programmation dynamique pour la variante entière du problème du sac-à-dos (exercice 16.2.2) est-il un algorithme à temps polynomial ? Pourquoi ?

34.1.5 Montrer qu'un algorithme à temps polynomial qui effectue au plus un nombre constant d'appels à des sous-programmes à temps polynomial s'exécute lui-même en temps polynomial, mais qu'un nombre polynomial d'appel à des sous-programmes à temps polynomial peut donner un algorithme à temps exponentiel.

34.1.6 Montrer que la classe P, vue en tant qu'ensemble de langages, est fermée pour l'union, l'intersection, la concaténation, le complément et l'opération Kleene étoile. Autrement dit, si $L_1, L_2 \in$ P, alors $L_1 \cup L_2 \in$ P, etc.

34.2 VÉRIFICATION EN TEMPS POLYNOMIAL

Nous nous intéressons maintenant aux algorithmes qui « vérifient » l'appartenance à un langage. Par exemple, supposons que pour une instance $\langle G, u, v, k \rangle$ donnée du problème de décision CHEMIN, on se donne aussi un chemin p de u vers v . On peut facilement vérifier si la longueur de p est au plus égale à k , et si oui, on peut voir p comme une « certification » de l'appartenance effective de l'instance à CHEMIN. Pour le problème de décision CHEMIN, cette certification ne semble pas si avantageuse que ça. Après tout, CHEMIN appartient à P (en fait, CHEMIN peut être résolu en temps linéaire) et la validation de l'appartenance à partir d'une certification prend autant de temps que la résolution du problème à partir de rien. Nous allons à présent examiner un problème pour lequel on ne connaît aucun algorithme de décision polynomial, bien que la validation d'une certification donnée soit aisée.

a) Cycles hamiltoniens

Le problème consistant à trouver un cycle hamiltonien dans un graphe non orienté est étudié depuis plus d'un siècle. Formellement, un *cycle hamiltonien* d'un graphe non orienté $G = (S, A)$ est un cycle élémentaire qui contient chaque sommet de S . Un graphe qui contient un cycle hamiltonien est dit *hamiltonien* ; sinon il est *non*

hamiltonien. Bondy et Murty [45] citent une lettre de W. R. Hamilton décrivant un jeu mathématique concernant le dodécaèdre (figure 34.2(a)) où un joueur plante cinq épingles sur cinq sommets consécutifs quelconques, et où l'autre joueur doit compléter le chemin pour former un cycle contenant tous les sommets. Le dodécaèdre est hamiltonien, et la figure 34.2(a) montre un cycle hamiltonien. Toutefois, tous les graphes ne sont pas hamiltoniens. Par exemple, la figure 34.2(b) montre un graphe biparti avec un nombre de sommets impair. (L'exercice 34.2.2 demande de montrer que tous les graphes de ce type sont non hamiltoniens.)

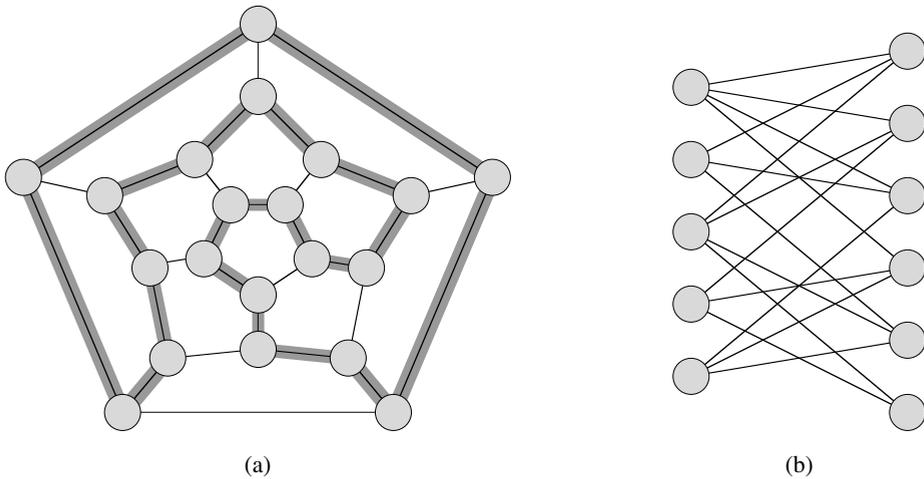


Figure 34.2 (a) Un graphe représentant les sommets, arêtes et faces d'un dodécaèdre, avec un cycle hamiltonien matérialisé par les arêtes ombrées. (b) Un graphe biparti avec un nombre impair de sommets. Un tel graphe est non hamiltonien.

On peut définir le **problème du cycle hamiltonien**, « Le graphe G possède-t-il un cycle hamiltonien ? » comme un langage formel :

$$\text{CYCLE-HAM} = \{ \langle G \rangle : G \text{ est un graphe hamiltonien} \} .$$

Comment un algorithme pourrait-il décider du langage CYCLE-HAM ? Étant donnée une instance $\langle G \rangle$ du problème, un algorithme de décision possible liste toutes les permutations des sommets de G , puis teste chaque permutation pour voir si elle forme un chemin hamiltonien. Quel est le temps d'exécution de cet algorithme ? Si l'on utilise l'encodage « raisonnable » d'un graphe par sa matrice d'adjacence, le nombre m de sommets du graphe est $\Omega(\sqrt{n})$, où $n = |\langle G \rangle|$ est la longueur de l'encodage de G . Il existe $m!$ permutations possible des sommets, et le temps d'exécution est donc $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, ce qui ne peut pas être mis sous la forme $O(n^k)$, quelle que soit la constante k . Cet algorithme naïf ne s'exécute donc pas en temps polynomial. En fait, le problème du cycle hamiltonien est NP-complet, ce qui sera démontré à la section 34.5.

b) Algorithmes de vérification

Considérons un problème légèrement plus simple. Supposons qu'un ami vous assure qu'un graphe G est hamiltonien, et qu'il se propose de vous le démontrer en vous donnant les sommets dans l'ordre du cycle hamiltonien. La démonstration sera certainement assez facile à valider : il suffit de regarder si le cycle proposé est bien hamiltonien en vérifiant que c'est une permutation des sommets de S et que chacune des arêtes consécutives du cycle se trouve effectivement dans le graphe. Cet algorithme de validation peut à coup sûr être implémenté pour s'exécuter en $O(n^2)$, où n est la longueur de l'encodage de G . Donc, une démonstration de l'existence d'un cycle hamiltonien dans un graphe peut être validée en temps polynomial.

On définit un **algorithme de vérification** comme un algorithme A à deux arguments, où un argument est une chaîne ordinaire x et l'autre une chaîne binaire y appelée **certificat**. Un algorithme A à deux arguments **vérifie** une chaîne x s'il existe un certificat y tel que $A(x, y) = 1$. Le **langage vérifié** par un algorithme de vérification A est

$$L = \{x \in \{0, 1\}^* : \text{il existe } y \in \{0, 1\}^* \text{ tel que } A(x, y) = 1\} .$$

Intuitivement, un algorithme A vérifie un langage L si, pour toute chaîne $x \in L$, il existe un certificat y que A peut utiliser pour démontrer que $x \in L$. Par ailleurs, pour toute chaîne $x \notin L$, il ne doit y avoir aucun certificat prouvant que $x \in L$. Par exemple, dans le problème du cycle hamiltonien, le certificat est la liste des sommets du cycle hamiltonien. Si un graphe est hamiltonien, le cycle hamiltonien lui-même offre assez d'information pour vérifier ce fait. Réciproquement, si un graphe n'est pas hamiltonien, il n'existe aucune liste de sommets capable de faire croire à l'algorithme de vérification que le graphe est hamiltonien, puisque l'algorithme contrôle soigneusement la validité du « cycle » proposé.

c) Classe de complexité NP

La **classe de complexité NP** est la classe des langages pouvant être validés par un algorithme polynomial ⁽⁶⁾. Plus précisément, un langage L appartient à NP si et seulement si il existe un algorithme polynomial A à deux entrées et une constante c telle que

$$L = \{x \in \{0, 1\}^* : \text{il existe un certificat } y \text{ avec } |y| = O(|x|^c) \\ \text{tel que } A(x, y) = 1\} .$$

On dit que l'algorithme A **vérifie** le langage L **en temps polynomial**.

On peut inférer de notre précédente discussion sur le problème du cycle hamiltonien que CYCLE-HAM \in NP. (Il est toujours agréable de savoir qu'un ensemble

(6) Le nom « NP » signifie « Non déterministe Polynomial ». La classe NP fut étudiée à l'origine dans le contexte du non déterminisme, mais ce livre utilise une notion plus simple, bien qu'équivalente, de vérification. Hopcroft et Ullman [156] donnent une bonne présentation de la NP-complétude en termes de modèles de calcul non déterministes.

important n'est pas vide.) De plus, si $L \in P$, alors $L \in NP$, puisque s'il existe un algorithme polynomial pouvant décider de L , l'algorithme peut être facilement converti en un algorithme de vérification à deux arguments qui se contente d'ignorer le certificat et accepte exactement les chaînes d'entrée dont il détermine qu'elle appartient à L . Donc, $P \subseteq NP$.

On ne sait pas si $P = NP$, mais la plupart des chercheurs pensent que P et NP sont deux classes différentes. Intuitivement, la classe P est composée de problèmes qui peuvent être résolus rapidement. La classe NP est composée de problèmes pour lesquels une solution peut être rapidement vérifiée. Vous avez peut-être expérimenté qu'il était souvent plus difficile de résoudre un problème à partir de zéro que de valider une solution clairement présentée, particulièrement quand les contraintes de temps sont importantes. Les théoriciens de l'informatique pensent en général que cette analogie s'étend aux classes P et NP , et donc que NP inclut des langages qui ne sont pas dans P .

Autre indice, encore plus flagrant, que $P \neq NP$: il existe des langages « NP-complets ». Nous allons étudier cette classe à la section 34.3.

Au-delà de la question $P \neq NP$, beaucoup d'autres questions fondamentales restent encore sans réponse. Malgré le travail important effectué par de nombreux chercheurs, personne n'est même capable de dire si la classe NP est fermée pour l'opération complément. Autrement dit, a-t-on $L \in NP$ implique $\bar{L} \in NP$? On peut définir la **classe de complexité co-NP** comme l'ensemble des langages L tels que $\bar{L} \in NP$. La question de savoir si NP est fermé pour le complément peut être réexprimée sous la forme $NP = \text{co-NP}$. Puisque P est fermé pour le complément (exercice 34.1.6), il s'ensuit que $P \subseteq NP \cap \text{co-NP}$. Une fois encore, on ne sait pas si $P = NP \cap \text{co-NP}$ ou s'il existe un langage dans $NP \cap \text{co-NP} - P$. La figure 34.3 montre les quatre scénarios possibles.

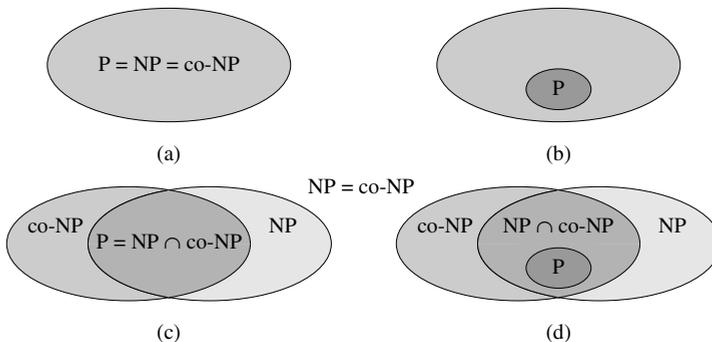


Figure 34.3 Quatre possibilités de relations entre les classes de complexité. Dans chaque diagramme, un région renfermant une autre indique une relation de sous-ensemble propre. (a) $P = NP = \text{co-NP}$. La plupart des chercheurs considèrent cette possibilité comme étant la plus improbable. (b) Si NP est fermé pour le complément, alors $NP = \text{co-NP}$, mais il n'est pas nécessaire que $P = NP$. (c) $P = NP \cap \text{co-NP}$, mais NP n'est pas fermé pour le complément. (d) $NP \neq \text{co-NP}$ et $P \neq NP \cap \text{co-NP}$. La plupart des chercheurs considère cette possibilité comme étant la plus probable.

Notre compréhension de la relation précise entre P et NP est donc malheureusement incomplète. Néanmoins, en explorant la théorie de la NP-complétude, nous allons voir que notre incapacité à démontrer qu'un problème est intraitable est, d'un point de vue pratique, loin d'être aussi grave qu'on pourrait le supposer.

Exercices

34.2.1 Soit le langage GRAPHES-ISOMORPHES = $\{\langle G_1, G_2 \rangle : G_1 \text{ et } G_2 \text{ sont des graphes isomorphes}\}$. Démontrer que GRAPHES-ISOMORPHES \in NP en décrivant un algorithme à temps polynomial qui vérifie le langage.

34.2.2 Démontrer que, si G est un graphe biparti non orienté comportant un nombre impair de sommets, alors G est non hamiltonien.

34.2.3 Montrer que, si CYCLE-HAM \in P, alors le problème consistant à donner la liste des sommets d'un cycle hamiltonien dans l'ordre, peut se résoudre en temps polynomial.

34.2.4 Démontrer que la classe NP des langages est fermée pour l'union, l'intersection, la concaténation et l'opération Kleene étoile. Étudier la fermeture de NP pour l'opération complément.

34.2.5 Montrer qu'un langage quelconque de NP peut être décidé par un algorithme s'exécutant en $2^{O(n^k)}$ pour une certaine constante k .

34.2.6 Un *chemin hamiltonien* dans un graphe est un chemin élémentaire qui visite chaque sommet exactement une fois. Montrer que le langage CHEMIN-HAM = $\{\langle G, u, v \rangle : \text{il existe un chemin hamiltonien de } u \text{ vers } v \text{ dans le graphe } G\}$ appartient à NP.

34.2.7 Montrer que le problème du chemin hamiltonien peut être résolu en temps polynomial sur les graphes orientés sans circuits. Donner un algorithme efficace pour le problème.

34.2.8 Soit ϕ une formule booléenne construite à partir de variables booléennes x_1, x_2, \dots, x_k , de négations (\neg), de ET (\wedge), de OU (\vee) et de parenthèses. La formule ϕ est une *tautologie* si elle donne 1 pour toute assignation de 1 et de 0 aux variables d'entrée. On définit TAUTOLOGIE comme le langage des formules booléennes qui sont des tautologies. Montrer que TAUTOLOGIE \in co-NP.

34.2.9 Démontrer que P \subseteq co-NP.

34.2.10 Démontrer que, si NP \neq co-NP, alors P \neq NP.

34.2.11 Soit G un graphe non orienté connexe, comportant au moins 3 sommets, et soit G^3 le graphe obtenu en reliant toutes les paires de sommets qui sont reliées par un chemin de G de longueur au plus égale à 3. Démontrer que G^3 est hamiltonien. (*conseil* : Construire un arbre couvrant pour G , et utiliser un raisonnement par récurrence.)

34.3 NP-COMPLÉTUDE ET RÉDUCTIBILITÉ

La raison qui pousse le plus les théoriciens de l'informatique à croire que $P \neq NP$ est sans doute l'existence de la classe des problèmes « NP-complets ». Cette classe possède une propriété surprenante : si *un* problème NP-complet peut être résolu en temps polynomial, alors *tous* les problèmes de NP peuvent être résolus en temps polynomial, autrement dit, $P = NP$. Pourtant, malgré des années de recherches, aucun algorithme polynomial n'a jamais été découvert pour quelque problème NP-complet que ce soit.

Décider du langage CYCLE-HAM est un problème NP-complet. Si l'on pouvait décider CYCLE-HAM en temps polynomial, on pourrait résoudre tous les problèmes de NP en temps polynomial. En fait, si $NP = P$ devait s'avérer non vide, on pourrait affirmer que $CYCLE-HAM \in NP = P$.

Les langages NP-complets sont, dans un sens, les langages les plus « difficiles » de NP. Dans cette section, nous allons montrer comment comparer la « difficulté » relative des langages à l'aide d'une notion précise appelée « réductibilité en temps polynomial ». Pour commencer, on définit formellement les langages NP-complets, puis on esquisse une preuve de la NP-complétude de l'un de ces langages, appelé CIRCUIT-SAT. A la section 34.5, on utilisera la notion de réductibilité pour montrer que beaucoup d'autres problèmes sont NP-complets.

a) Réductibilité

Intuitivement, un problème Q peut être ramené à un autre problème Q' si une instance quelconque de Q peut être « facilement reformulée » comme une instance de Q' , dont la solution fournira une solution pour l'instance de Q . Par exemple, le problème de la résolution d'équations linéaires à une inconnue x peut se réduire au problème de la résolution d'équations quadratiques. Étant donnée une instance $ax + b = 0$, on la transforme en $0x^2 + ax + b = 0$, dont la solution fournit une solution à $ax + b = 0$. Donc, si un problème Q se réduit à un autre problème Q' , alors Q n'est, dans un sens, « pas plus difficile à résoudre » que Q' .

En revenant à notre cadre des langages formels pour les problèmes de décision, on dit qu'un langage L_1 est **réductible en temps polynomial** à un langage L_2 , ce qui s'écrit $L_1 \leq_P L_2$, s'il existe une fonction calculable en temps polynomial $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ telle que pour tout $x \in \{0, 1\}^*$,

$$x \in L_1 \text{ si et seulement si } f(x) \in L_2. \quad (34.1)$$

On appelle la fonction f la **fonction de réduction**, et un algorithme polynomial F qui calcule f est appelé **algorithme de réduction**.

La figure 34.4 illustre le principe d'une réduction en temps polynomial d'un langage L_1 à un autre langage L_2 . Chaque langage est un sous-ensemble de $\{0, 1\}^*$. La

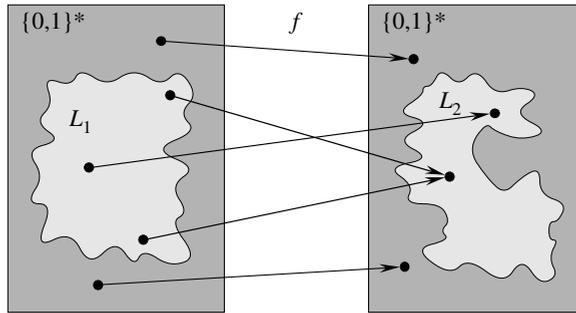


Figure 34.4 Une illustration d’une réduction en temps polynomial d’un langage L_1 à un langage L_2 via une fonction de réduction f . Pour une entrée $x \in \{0, 1\}^*$ quelconque, la question de savoir si $x \in L_1$ a la même réponse que la question de savoir si $f(x) \in L_2$.

fonction de réduction f fournit une correspondance à temps polynomial telle que si $x \in L_1$, alors $f(x) \in L_2$. De plus, si $x \notin L_1$, alors $f(x) \notin L_2$. Donc, la fonction de réduction fait correspondre une instance x quelconque du problème de décision représenté par le langage L_1 à une instance $f(x)$ du problème représenté par L_2 . L’obtention d’une réponse à la question $f(x) \in L_2$ fournit directement une réponse à la question $x \in L_1$. Les réductions en temps polynomial nous donnent un outil puissant pour démontrer que divers langages appartiennent à P.

Lemme 34.3 Si $L_1, L_2 \subseteq \{0, 1\}^*$ sont des langages tels que $L_1 \leq_P L_2$, alors $L_2 \in P$ implique $L_1 \in P$.

Démonstration : Soit A_2 un algorithme polynomial qui décide de L_2 , et soit F un algorithme de réduction à temps polynomial qui calcule la fonction de réduction f . Nous allons construire un algorithme A_1 polynomial qui décide de L_1 .

La figure 34.5 illustre la construction de A_1 . Pour une entrée $x \in \{0, 1\}^*$ donnée, l’algorithme A_1 utilise F pour transformer x en $f(x)$, puis fait appel à A_2 pour tester si $f(x) \in L_2$. Le résultat de A_2 sera la valeur fournie comme résultat par A_1 .

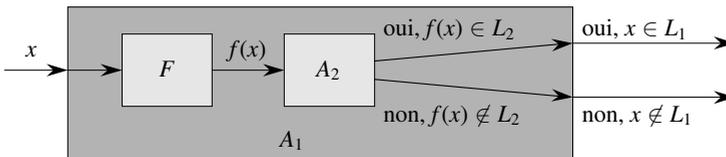


Figure 34.5 La démonstration du lemme 34.3. L’algorithme F est un algorithme de réduction qui calcule la fonction de réduction f de L_1 à L_2 en temps polynomial, et A_2 est un algorithme polynomial qui décide de L_2 . On peut voir si l’algorithme A_1 est capable de décider si $x \in L_1$ en utilisant F pour transformer une entrée x quelconque en $f(x)$, puis en utilisant A_2 pour décider si $f(x) \in L_2$.

La validité de A_1 se déduit de la condition (34.1). L'algorithme s'exécute en temps polynomial, puisque F et A_2 s'exécutent en temps polynomial (voir exercice 34.1.5). \square

b) NP-complétude

Les réductions à temps polynomial fournissent un moyen formel de montrer qu'un problème est au moins aussi difficile qu'un autre, à un facteur temporel polynomial près. Autrement dit, si $L_1 \leq_P L_2$, alors L_1 n'est pas plus difficile (à un facteur polynomial près) que L_2 , ce qui explique le choix de la notation « inférieur ou égal » pour la réduction. On peut à présent définir l'ensemble des langages NP-complets, qui sont les problèmes les plus difficiles de NP.

Un langage $L \subseteq \{0, 1\}^*$ est **NP-complet** si

- 1) $L \in \text{NP}$, et
- 2) $L' \leq_P L$ pour tout $L' \in \text{NP}$.

Si un langage L vérifie la propriété 2, mais pas forcément la propriété 1, on dit que L est **NP-difficile**. On définit également NPC comme la classe des langages NP-complets.

Comme le montre le théorème suivant, la NP-complétude est une notion cruciale pour la question de savoir si P est en fait égal à NP.

Théorème 34.4 *S'il existe un problème NP-complet qui est résoluble en temps polynomial, alors $P = \text{NP}$. De façon équivalente, s'il existe un problème de NP qui n'est pas résoluble en temps polynomial, alors aucun problème NP-complet n'est résoluble en temps polynomial.*

Démonstration : Supposons que $L \in P$ et aussi que $L \in \text{NPC}$. Pour tout $L' \in \text{NP}$, on a $L' \leq_P L$ d'après la propriété 2 de la définition de la NP-complétude. Donc, d'après le lemme 34.3, on a également $L' \in P$, ce qui démontre la première assertion du théorème.

Pour démontrer la seconde assertion, on remarque qu'elle est la contraposée de la première. \square

C'est pour cette raison que la recherche autour de la question $P \neq \text{NP}$ est centrée sur les problèmes NP-complets. La plupart des théoriciens pensent que $P \neq \text{NP}$, ce qui conduit aux relations décrites à la figure 34.6 entre P, NP, et NPC. Mais pour le moment, il n'est pas exclu que quelqu'un fournisse un jour un algorithme polynomial pour un problème NP-complet, démontrant du même coup que $P = \text{NP}$. Néanmoins, comme aucun algorithme de ce type n'a encore été découvert pour quelque problème NP-complet que ce soit, démontrer qu'un problème est NP-complet donne un excellent indice de son intraitabilité.

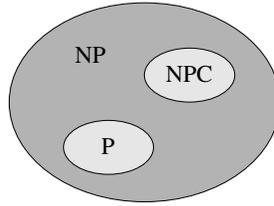


Figure 34.6 Comment la plupart des théoriciens de l'informatique voient les relations entre P, NP et NPC. P et NPC sont tous deux entièrement contenus dans NP, et $P \cap NPC = \emptyset$.

c) Satisfaisabilité d'un circuit

Nous avons défini la notion de problème NP-complet mais, jusqu'à présent, nous n'avons pas exhibé de problème qui soit NP-complet. Une fois que nous aurons montré qu'il existe au moins un problème NP-complet, nous pourrons utiliser la réductibilité à temps polynomial comme outil pour prouver la NP-complétude d'autres problèmes. Nous nous attachons donc maintenant à démontrer l'existence d'un problème NP-complet : le problème de satisfaisabilité d'un circuit.

Malheureusement, la preuve formelle du fait que le problème de satisfaisabilité d'un circuit est NP-complet fait appel à des détails techniques qui dépassent le champ de ce livre. En revanche, nous allons donner une démonstration informelle qui s'appuie sur une compréhension élémentaire des circuits booléens combinatoires.

Les circuits combinatoires booléens sont construits à partir d'éléments combinatoires reliés par des câbles. Un **élément combinatoire booléen** est un élément de circuit quelconque qui possède un nombre d'entrées et de sorties constant, et qui réalise une fonction bien définie. Les valeurs booléennes appartiennent à l'ensemble $\{0, 1\}$, où 0 représente FAUX et 1 représente VRAI. Les éléments combinatoires booléens que nous utiliserons dans le problème de la satisfaisabilité de circuit calculent une fonction booléenne simple, et on les appelle des **portes logiques**. La figure 34.7 montre les trois portes logiques de base qui seront utilisées dans le problème de la satisfaisabilité de circuit : la **porte NON** (ou **inverseur**), la **porte ET** et la **porte OU**. La porte NON prend une **entrée** binaire x unique, qui vaut 0 ou 1, et produit une **sortie** binaire z dont la valeur est l'opposée de celle de l'entrée. Chacune des deux autres portes prend deux entrées binaires x et y et produit une sortie binaire z unique.

Le fonctionnement de chaque porte, et d'un élément combinatoire booléen quelconque, peut être décrit par une **table de vérité**, représentée sous chaque porte de la figure 34.7. Une table de vérité donne les sorties de l'élément combinatoire pour chaque entrée possible. Par exemple, la table de vérité de la porte OU nous indique que l'entrée $x = 0$ et $y = 1$ engendre la sortie $z = 1$. On utilise les symboles \neg pour noter la fonction NON, \wedge pour noter la fonction ET et \vee pour noter la fonction OU. On écrira par exemple, $0 \vee 1 = 1$.

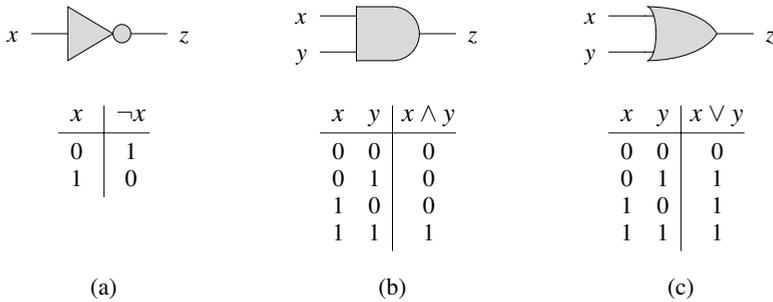


Figure 34.7 Trois portes logiques élémentaires, avec entrées et sorties binaires. Sous chaque porte figure la table de vérité qui décrit le fonctionnement de la porte. (a) La porte NON. (b) La porte ET. (c) La porte OU.

On peut généraliser les portes ET et OU pour qu'elles prennent plus de deux entrées. La sortie d'une porte ET est 1 si toutes les entrées sont des 1, et 0 sinon. La sortie d'une porte OU est 1 si l'une au moins des entrées est 1, et 0 sinon.

Un **circuit combinatoire booléen** se compose d'un ou plusieurs éléments combinatoires booléens interconnectés par des **fils** . Un fil peut relier la sortie d'un élément à l'entrée d'un autre ; ce faisant, la valeur en sortie du premier élément devient la valeur en entrée du second. La figure 34.8 montre deux circuits combinatoires booléens similaires, qui ne diffèrent que par une porte. La partie (a) de la figure montre aussi les valeurs des différents fils, étant donnée l'entrée $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$. Un même fil ne peut pas servir de sortie à plus d'un élément combinatoire, mais il peut servir d'entrée à plusieurs éléments. Le nombre d'éléments alimentés par un fil s'appelle l' **éventail (fan-out)** du fil. Si aucune sortie d'élément n'est reliée à un fil, celui-ci est alors une **entrée de circuit** , acceptant des valeurs d'entrée qui viennent d'une source externe. Si aucune entrée d'élément n'est reliée à un fil, celui-ci est alors une **sortie de circuit** , fournissant au monde extérieur les résultats des calculs faits par le circuit. (Un fil interne peut aussi être relié en éventail à une sortie de circuit.) Pour la définition du problème de la satisfaisabilité de circuit, on limitera le nombre de sorties de circuit à 1, alors que les matériels modernes permettent d'avoir des circuits combinatoires booléens à sorties multiples.

Les circuits combinatoires booléens n'ont pas de circuits. Autrement dit, supposez que l'on crée un graphe orienté $G = (S, A)$ ayant un sommet pour chaque élément combinatoire et k arcs pour chaque fil dont le degré d'éventail est k ; il y a un arc (u, v) s'il y a un fil qui relie la sortie de l'élément u à une entrée de l'élément v . Alors, G est forcément sans circuit.

Une **assignation de vérité** pour un circuit combinatoire booléen est un ensemble de valeurs d'entrée booléennes. On dit qu'un circuit combinatoire booléen à sortie unique est **satisfaisable** s'il a une **assignation satisfaisante** : une assignation de vérité qui entraîne que la sortie du circuit soit 1. Par exemple, le circuit de la figure 34.8(a) a l'assignation satisfaisante $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$, et donc il est satisfaisable. Comme

l'exercice 34.3.1 vous demandera de le montrer, aucune assignation de valeurs à x_1 , x_2 et x_3 ne peut amener le circuit de la figure 34.8(b) à produire 1 en sortie ; il produit toujours 0, et donc il est non satisfaisable.

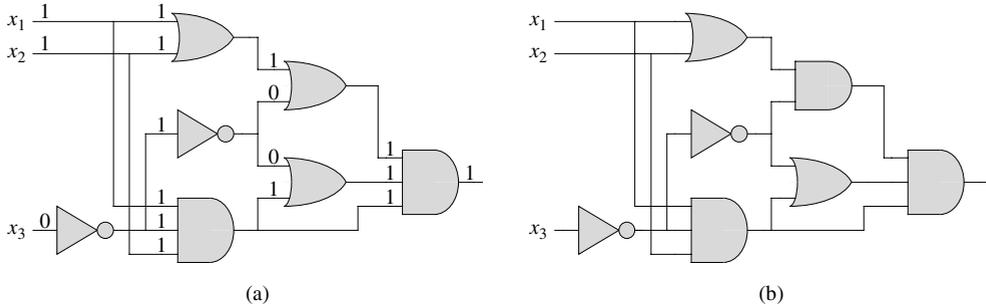


Figure 34.8 Deux instances du problème de la satisfaisabilité de circuit. (a) L'assignation $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ aux entrées de ce circuit engendre l'apparition en sortie d'un 1. Le circuit est donc satisfaisable. (b) Aucune assignation aux entrées de ce circuit ne provoque l'apparition en sortie d'un 1. Le circuit est donc non satisfaisable.

Le **problème de la satisfaisabilité d'un circuit** est « Étant donné un circuit combinatoire booléen composé de portes ET, OU et NON, est-il satisfaisable ? » Pour poser cette question de manière formelle, il faut toutefois se mettre d'accord sur un encodage standard pour les circuits. La **taille** d'un circuit combinatoire booléen est le nombre d'éléments combinatoires booléens, augmenté du nombre de fils. On peut élaborer un encodage inspiré de celui des graphes qui fait correspondre à un circuit C donné une chaîne binaire $\langle C \rangle$ dont la longueur est polynomiale en fonction de la taille du circuit lui-même. En tant que langage formel, on peut donc définir

CIRCUIT-SAT =

$$\{ \langle C \rangle : C \text{ est un circuit combinatoire booléen satisfaisable} \} .$$

Le problème de la satisfaisabilité de circuit a une grande importance dans le domaine de l'optimisation de circuits électroniques. Si un sous-circuit produit toujours la valeur 0, il peut être remplacé par un sous-circuit plus simple qui omet toutes les portes logiques et fournit la valeur constante 0 en sortie. Il serait avantageux d'avoir un algorithme polynomial pour ce problème.

Étant donné un circuit C , on pourrait essayer de déterminer s'il est satisfaisable en se contentant de tester toutes les assignations d'entrée possibles. Malheureusement, s'il existe k entrées, il y a 2^k affectations possibles. Quand la taille de C est polynomiale en k , tester chaque assignation prend un temps $\Omega(2^k)$, ce qui conduit à un

algorithme suprapolynomial par rapport à la taille du circuit. ⁽⁷⁾ En fait, comme cela a été affirmé, il y a des indices très forts qu'il n'existe aucun algorithme polynomial qui puisse résoudre le problème de la satisfaisabilité de circuit, car la satisfaisabilité de circuit est NP-complet. On divisera la démonstration en deux parties, correspondant aux deux parties de la définition de la NP-complétude.

Lemme 34.5 *Le problème de la satisfaisabilité de circuit appartient à la classe NP.*

Démonstration : Nous allons proposer un algorithme polynomial A à deux entrées, capable de vérifier CIRCUIT-SAT. L'une des entrées de A est (un encodage standard d') un circuit combinatoire booléen C . L'autre entrée est un certificat correspondant à une assignation de valeurs booléennes aux fils de C . (Voir exercice 34.3.4 pour un plus petit certificat.)

L'algorithme A est construit de la manière suivante. Pour chaque porte logique du circuit, il vérifie que la valeur fournie par le certificat sur le fil de sortie est correctement calculée comme fonction des valeurs présentées sur les fils d'entrée. Ensuite, si la sortie du circuit tout entier vaut 1, l'algorithme affiche 1, puisque les valeurs affectées aux entrées de C fournissent une assignation satisfaisante. Sinon, A affiche la valeur 0.

Chaque fois qu'un circuit satisfaisable C est fourni en entrée à l'algorithme A , il existe un certificat dont la longueur est polynomiale par rapport à la taille de C et qui provoque la sortie par A de la valeur 1. Chaque fois qu'un circuit non satisfaisable est fourni en entrée, aucun certificat ne peut faire croire à A que le circuit est satisfaisable. L'algorithme A s'exécute en temps polynomial : avec une bonne implémentation, on peut atteindre un temps linéaire. Donc, CIRCUIT-SAT peut être vérifié en temps polynomial et CIRCUIT-SAT \in NP. \square

La seconde partie de la preuve de la NP-complétude de CIRCUIT-SAT consiste à démontrer que le langage est NP-difficile. Autrement dit, on doit montrer que tout langage de NP est réductible en temps polynomial à CIRCUIT-SAT. La véritable démonstration de cette assertion est pleine de complications techniques, et nous allons donc nous contenter d'une ébauche de preuve, basée sur la compréhension de certains mécanismes matériels d'un ordinateur.

Un programme informatique est stocké dans la mémoire d'un ordinateur sous forme d'une séquence d'instructions. Une instruction typique encode une opération à effectuer, des adresses mémoire d'opérandes et une adresse où il faudra stocker le résultat. Un emplacement particulier de mémoire, appelé *pointeur d'instruction*, pointe constamment vers la prochaine instruction à exécuter. Le pointeur d'instruction est incrémenté automatiquement chaque fois qu'une instruction est exécutée, forçant ainsi l'ordinateur à exécuter les instructions séquentiellement. Cependant, une instruction peut modifier la valeur du pointeur d'instruction et altérer l'exécution

(7) En revanche, si la taille du circuit C est $\Theta(2^k)$, alors un algorithme dont le temps d'exécution est $O(2^k)$ a un temps d'exécution qui est polynomial par rapport à la taille du circuit. Même si $P \neq NP$, cette situation ne contredit pas le fait que le problème est NP-complet ; l'existence d'un algorithme à temps polynomial pour un cas particulier n'implique pas qu'il existe un algorithme à temps polynomial pour tous les cas.

séquentielle normale ; cela permet de programmer des boucles et des branchements conditionnels.

A n'importe quel instant de l'exécution d'un programme, l'état du calcul est représenté dans la mémoire de l'ordinateur. (On considère que la mémoire contient le programme lui-même, le pointeur d'instruction, l'espace de travail, plus les différents bits d'état gérés par l'ordinateur à des fins de gestion interne). Un état particulier de la mémoire est dit **configuration**. L'exécution d'une instruction peut être vue comme la mise en correspondance de deux configurations. Il est important de noter que le matériel qui accomplit cette mise en correspondance peut être implémenté par un circuit combinatoire booléen, que nous noterons M dans la démonstration du lemme suivant.

Lemme 34.6 *Le problème de la satisfaisabilité de circuit est NP-difficile.*

Démonstration : Soit L un langage de NP. Nous allons décrire un algorithme polynomial F qui calcule une fonction de réduction f faisant correspondre toute chaîne binaire x à un circuit $C = f(x)$ tel que $x \in L$ si et seulement si $C \in \text{CIRCUIT-SAT}$.

Puisque $L \in \text{NP}$, il doit exister un algorithme A qui vérifie L en temps polynomial. L'algorithme F que nous allons construire utilisera l'algorithme A à deux entrées pour calculer la fonction de réduction f .

Soit $T(n)$ le temps d'exécution le plus défavorable de l'algorithme A sur des chaînes de longueur n , et soit $k \geq 1$ une constante telle que $T(n) = O(n^k)$ et que la longueur du certificat soit $O(n^k)$. (Le temps d'exécution de A est, en réalité, une fonction polynomiale de la taille d'entrée totale, ce qui inclut à la fois une chaîne d'entrée et un certificat ; mais la longueur du certificat étant polynomiale par rapport à la longueur n de la chaîne d'entrée, le temps d'exécution est polynomial en n .)

Le principe premier de cette démonstration est de représenter le calcul effectué par A comme une séquence de configurations. Comme on peut le voir à la figure 34.9, chaque configuration peut être divisée en plusieurs parties : programme implémentant A , pointeur d'instruction, registres d'état, entrée x , certificat y et espace de travail. En commençant avec une configuration initiale c_0 , chaque configuration c_i est mise en correspondance avec la configuration suivante c_{i+1} par le circuit combinatoire M implémentant le matériel informatique. La sortie de l'algorithme, 0 ou 1, est écrite dans un emplacement spécifique de l'espace de travail à la fin de l'exécution de A et, si l'on suppose que A s'arrête après cela, cette valeur n'est jamais modifiée. Donc, si l'algorithme s'exécute pendant au plus $T(n)$ étapes, la sortie apparaît comme l'un des bits de $c_{T(n)}$.

L'algorithme de réduction F construit un circuit combinatoire unique qui calcule toutes les configurations produites par une configuration initiale donnée. L'idée est d'assembler $T(n)$ copies du circuit M . La sortie du i ème circuit, qui produit la configuration c_i , alimente directement l'entrée du $(i + 1)$ ème circuit. Donc, les configurations, au lieu de finir dans un registre d'état, prennent simplement la forme de valeurs sur les fils reliant des copies de M .

N'oublions pas la tâche dévolue à l'algorithme F de réduction en temps polynomial. Étant donnée une entrée x , il doit calculer un circuit $C = f(x)$ qui est satisfaisable si et seulement si il existe un certificat y tel que $A(x, y) = 1$. Lorsqu'on fournit à F

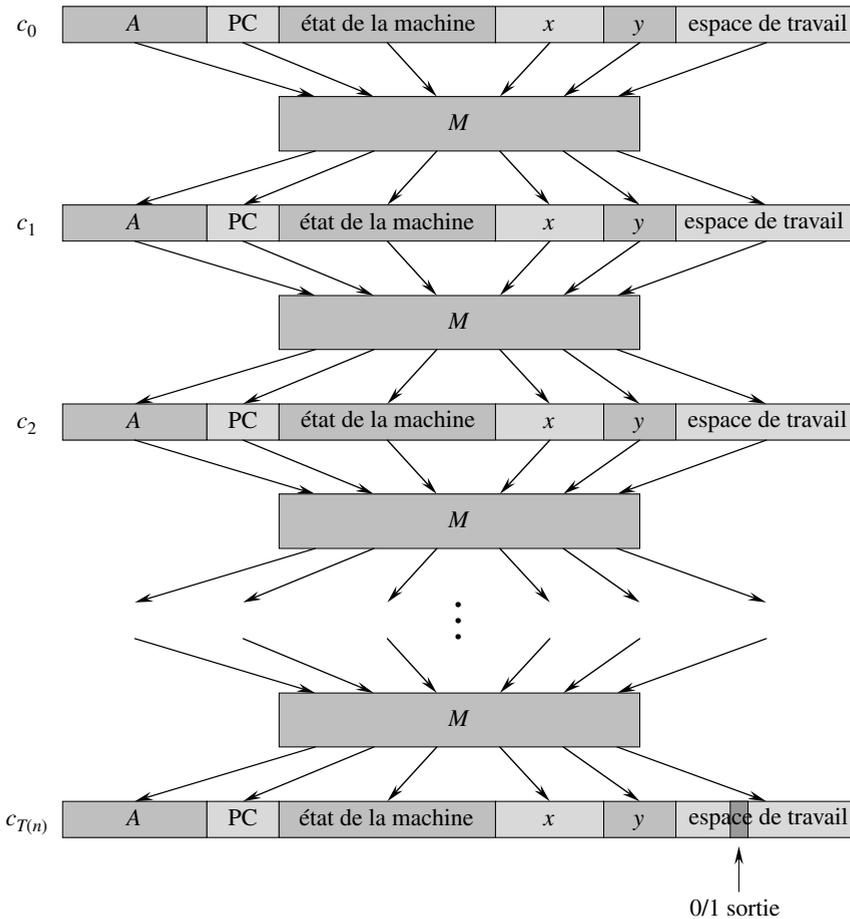


Figure 34.9 La séquence de configurations produites par un algorithme A s'exécutant sur une entrée x et un certificat y . Chaque configuration représente l'état de l'ordinateur à une étape du calcul et, en plus de A , x et y , inclut le pointeur d'instruction (PC), les registres d'état, plus l'espace de travail. Excepté pour le certificat y , la configuration initiale c_0 est constante. Chaque configuration est associée à la suivante par un circuit combinatoire booléen M . La sortie est un bit distingué de l'espace de travail.

une entrée x , il commence par calculer $n = |x|$ et construit un circuit combinatoire C' constitué de $T(n)$ copies de M . L'entrée fournie à C' est une configuration initiale correspondant à un calcul sur $A(x, y)$, et la sortie est la configuration $c_{T(n)}$.

Le circuit $C = f(x)$ calculé par F est obtenu en modifiant légèrement C' . Primo, les entrées vers C' correspondant au programme implémentant A , au pointeur d'instruction initial, à l'entrée x et à l'état initial de la mémoire sont câblées directement vers ces valeurs connues. Donc, les seules entrées restantes du circuit correspondent au certificat y . Ensuite, toutes les sorties vers le circuit sont ignorées, hormis le bit de $c_{T(n)}$ correspondant à la sortie de A . Ce circuit C ainsi construit calcule $C(y) = A(x, y)$ pour

une entrée y quelconque de longueur $O(n^k)$. L'algorithme de réduction F , quand on lui fournit une chaîne x en entrée, calcule un tel circuit et le produit en sortie.

Deux propriétés restent à démontrer. D'abord, on doit montrer que F calcule correctement une fonction de réduction f . Autrement dit, on doit montrer que C est satisfaisable si et seulement si il existe un certificat y tel que $A(x, y) = 1$. Ensuite, il faut montrer que F s'exécute en temps polynomial.

Pour montrer que F calcule correctement une fonction de réduction, supposons qu'il existe un certificat y de longueur $O(n^k)$ tel que $A(x, y) = 1$. Alors, si l'on applique les bits de y aux entrées de C , la sortie de C est $C(y) = A(x, y) = 1$. Donc, si un certificat existe, C est satisfaisable. Pour la réciproque, supposons que C soit satisfaisable. Alors il existe une entrée y appliquée à C telle que $C(y) = 1$, d'où l'on conclut que $A(x, y) = 1$. Donc, F calcule correctement une fonction de réduction.

Pour compléter notre esquisse de démonstration, il nous suffit de montrer que F s'exécute en temps polynomial par rapport à $n = |x|$. La première observation est que le nombre de bits requis pour représenter une configuration est polynomial en n . Le programme pour A a lui-même une taille constante, indépendante de la longueur de son entrée x . La longueur de l'entrée x est n , et la longueur du certificat y est $O(n^k)$. Puisque l'algorithme s'exécute pendant au plus $O(n^k)$ étapes, la quantité d'espace de travail requise par A est également polynomiale en n . (On suppose que cette mémoire est contiguë ; l'exercice 34.3.5 demande d'étendre l'argumentation à la situation où les emplacements auxquels A accède sont dispersés sur une région de mémoire beaucoup plus grande et où le motif de dispersion peut varier d'une entrée x à l'autre.)

Le circuit combinatoire M implémentant le matériel informatique a une taille polynomiale par rapport à la longueur d'une configuration, qui est polynomiale en $O(n^k)$ et donc polynomiale en n . (La plus grande partie de ce montage implémente la logique du système de gestion de la mémoire.) Le circuit C est composé d'au plus $t = O(n^k)$ copies de M , et il a donc une taille polynomiale en n . La construction de C à partir de x peut être accomplie en temps polynomial par l'algorithme de réduction F , puisque chaque étape de la construction prend un temps polynomial. \square

Le langage CIRCUIT-SAT est donc au moins aussi difficile que n'importe quel langage de NP, et puisqu'il appartient à NP, il est NP-complet.

Théorème 34.7 *Le problème de la satisfaisabilité de circuit est NP-complet.*

Démonstration : Immédiate d'après les lemmes 34.5 et 34.6 et d'après la définition de la NP-complétude. \square

Exercices

34.3.1 Vérifier que le circuit de la figure 34.8(b) est non satisfaisable.

34.3.2 Montrer que la relation \leq_P est une relation transitive sur les langages. Autrement dit, montrer que si $L_1 \leq_P L_2$ et $L_2 \leq_P L_3$, alors $L_1 \leq_P L_3$.

34.3.3 Démontrer que $L \leq_P \bar{L}$ si et seulement si $\bar{L} \leq_P L$.

34.3.4 Montrer qu'une assignation satisfaisante peut être utilisée comme certificat dans une autre démonstration du lemme 34.5. Quel certificat produit une démonstration plus facile ?

34.3.5 La preuve du lemme 34.6 suppose que l'espace de travail de l'algorithme A occupe une région contiguë de taille polynomiale. A quel endroit de la démonstration cette hypothèse est-elle exploitée ? Montrer que cette hypothèse ne remet pas en cause le caractère général de la démonstration.

34.3.6 Un langage L est **complet** pour une classe de langage C par rapport aux réductions à temps polynomial si $L \in C$ et $L' \leq_P L$ pour tout $L' \in C$. Montrer que \emptyset et $\{0, 1\}^*$ sont les seuls langages de P qui ne sont pas complets pour P par rapport aux réductions à temps polynomial.

34.3.7 Montrer que L est complet pour NP si et seulement si \bar{L} est complet pour co-NP.

34.3.8 L'algorithme de réduction F dans la démonstration du lemme 34.6 construit le circuit $C = f(x)$ en se fondant sur la connaissance qu'il a de x , A et k . Le professeur Sartre observe que la chaîne x est une entrée de F , mais que seule l'existence de A , de k et du facteur constant implicite au temps d'exécution $O(n^k)$ est connue de F (puisque le langage L appartient à NP), pas leurs valeurs réelles. Le professeur en conclut donc que F ne peut pas construire le circuit C et que le langage CIRCUIT-SAT n'est pas forcément NP-difficile. Montrer la faille dans le raisonnement du professeur.

34.4 PREUVES DE NP-COMPLÉTUDE

La NP-complétude du problème de la satisfaisabilité d'un circuit s'appuie sur une démonstration directe que $L \leq_P \text{CIRCUIT-SAT}$ pour tout langage $L \in \text{NP}$. Dans cette section, on montrera comment démontrer que des langages sont NP-complets sans réduire directement *chaque* langage de NP au langage donné. Nous allons illustrer cette méthodologie en démontrant que divers problèmes de satisfaisabilité de formule sont NP-complets. La section 34.5 fournit de nombreux autres exemples de cette méthodologie.

Le lemme suivant constitue la fondation de notre méthode de démonstration qu'un langage est NP-complet.

Lemme 34.8 *Si L est un langage tel que $L' \leq_P L$ pour un certain $L' \in \text{NPC}$, alors L est NP-difficile. De plus, si $L \in \text{NP}$, alors $L \in \text{NPC}$.*

Démonstration : Puisque L' est NP-complet, pour tout $L'' \in \text{NP}$, on a $L'' \leq_P L'$. Par hypothèse, $L' \leq_P L$, et donc par transitivité (exercice 34.3.2), on a $L'' \leq_P L$, ce qui montre que L est NP-difficile. Si $L \in \text{NP}$, on a également $L \in \text{NPC}$. \square

Autrement dit, en réduisant un langage NP-complet L' connu à L , on réduit implicitement tout langage de NP à L . Donc, le lemme 34.8 nous donne une méthode pour démontrer qu'un langage L est NP-complet :

- 1) Prouver que $L \in \text{NP}$.
- 2) Choisir un langage L' NP-complet connu.
- 3) Décrire un algorithme qui calcule une fonction f faisant correspondre toute instance $x \in \{0, 1\}^*$ de L' à une instance $f(x)$ de L .
- 4) Démontrer que la fonction f satisfait $x \in L'$ si et seulement si $f(x) \in L$ pour tout $x \in \{0, 1\}^*$.
- 5) Démontrer que l'algorithme calculant f s'exécute en temps polynomial.

(Les étapes 2–5 montrent que L est NP-difficile.) Cette méthode de réduction à partir d'un même langage NP-complet connu est beaucoup plus simple que le procédé qui consiste à montrer directement comment réduire à partir de chaque langage de NP. Démontrer que CIRCUIT-SAT \in NPC nous a permis de coincer un « pied dans la porte ». A présent que nous savons que le problème de la satisfaisabilité de circuit est NP-complet, nous pouvons démontrer beaucoup plus facilement que d'autres problèmes sont NP-complets. Par ailleurs, plus nous enrichissons le catalogue des problèmes NP-complets connus, plus nous aurons de choix pour le langage à partir duquel on réduit.

a) Satisfaisabilité de formule

Nous illustrons la méthode de la réduction en donnant une preuve de la NP-complétude du problème consistant à déterminer si une formule booléenne, et non plus un circuit, est satisfaisable. Ce problème eut l'honneur d'être le premier dans l'histoire dont on a démontré qu'il était NP-complet.

Nous exprimons le problème de la *satisfaisabilité (de formule)* en fonction du langage SAT de la manière suivante. Une instance de SAT est une formule booléenne ϕ composée de

- 1) n variables booléennes : x_1, x_2, \dots, x_n ;
- 2) m connecteurs booléens : toute fonction booléenne avec une ou deux entrées et une sortie, telle que \wedge (ET), \vee (OU), \neg (NON), \rightarrow (implication), \leftrightarrow (si et seulement si) ; et
- 3) des parenthèses. (Sans nuire à la généralité, on suppose qu'il n'y a pas de parenthèses redondantes, c'est à dire qu'il y a au plus une paire de parenthèses par connecteur booléen.)

Il est facile d'encoder une formule booléenne ϕ dans une longueur qui est polynomiale en $n + m$. Comme c'est le cas avec les circuits combinatoires booléens, une *assignation de vérité* pour une formule booléenne ϕ est un ensemble de valeurs pour les variables de ϕ , et une *assignation satisfaisante* est une assignation pour laquelle l'évaluation de la formule donne 1. Une formule possédant une assignation satisfaisante est une formule *satisfaisable*. Le problème de la satisfaisabilité consiste à savoir si une formule booléenne donnée est satisfaisable ; en termes des langages formels,

on dira

$$\text{SAT} = \{ \langle \phi \rangle : \phi \text{ est une formule booléenne satisfaisable} \} .$$

A titre d'exemple, la formule

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

possède l'assignation satisfaisante $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$, puisque

$$\begin{aligned} \phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 & (34.2) \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 , \end{aligned}$$

et donc cette formule ϕ appartient à SAT.

L'algorithme naïf permettant de déterminer si une formule booléenne arbitraire est satisfaisable ne s'exécute pas en temps polynomial. Il existe 2^n assignations possibles d'une formule ϕ à n variables. Si la longueur de $\langle \phi \rangle$ est polynomiale en n , alors la vérification de chaque assignation demande un temps $\Omega(2^n)$, ce qui est suprapolynomial par rapport à la longueur de $\langle \phi \rangle$. Comme le montre le théorème suivant, un algorithme polynomial a très peu de chances d'exister.

Théorème 34.9 *La satisfaisabilité d'une formule booléenne est un problème NP-complet.*

Démonstration : Nous commencerons par montrer que $\text{SAT} \in \text{NP}$. Ensuite, nous montrerons que SAT est NP-difficile en montrant que $\text{CIRCUIT-SAT} \leq_P \text{SAT}$; d'après le lemme 34.8, cela prouvera le théorème.

Pour montrer que SAT appartient à NP, on montre qu'un certificat consistant en une assignation satisfaisante pour une formule ϕ en entrée peut être validé en temps polynomial. L'algorithme de vérification remplace tout simplement chaque variable de la formule par sa valeur correspondante, puis évalue l'expression d'une façon qui rappelle beaucoup ce que nous avons fait dans l'équation (34.2) précédente. Ce travail peut facilement s'effectuer en temps polynomial. Si l'expression a pour évaluation 1, la formule est satisfaisable. Donc, la première condition du lemme 34.8 pour la NP-complétude est vérifiée.

Pour démontrer que SAT est NP-difficile, on montre que $\text{CIRCUIT-SAT} \leq_P \text{SAT}$. Autrement dit, toute instance du problème de satisfaisabilité de circuit peut être réduite en temps polynomial à une instance du problème de satisfaisabilité de formule. On peut faire appel à une récurrence pour exprimer un circuit combinatoire booléen quelconque comme une formule booléenne. On regarde simplement la porte qui produit le résultat du circuit, et on exprime de manière récurrente chacune des entrées de la porte en tant que formules. La formule correspondant au circuit est alors obtenue en écrivant une expression qui applique la fonction de la porte aux formules de ses entrées.

Malheureusement, cette méthode directe n'est pas une réduction en temps polynomial. Les sous-formules partagées peuvent faire croître exponentiellement la taille de la formule générée, pour peu qu'il y ait des portes dont les fils de sortie aient des facteurs

d'éventail de 2 ou plus (voir exercice 34.4.1). Donc, l'algorithme de réduction devra être un peu plus astucieux.

La figure 34.10 illustre le principe fondamental de la réduction de CIRCUIT-SAT à SAT sur le circuit de la figure 34.8(a). Pour chaque fil x_i du circuit C , la formule ϕ a une variable x_i . L'opération propre à une porte peut maintenant être exprimée comme une formule mettant en jeu les variables situées de ses fils incidents. Par exemple, l'opération de la porte ET en sortie est $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

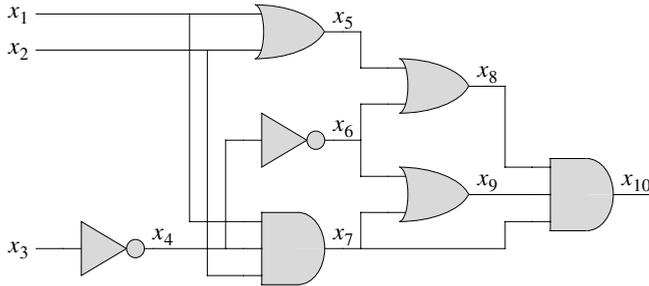


Figure 34.10 Réduction du problème de la satisfaisabilité de circuit à celui de la satisfaisabilité de formule. La formule produite par l'algorithme de réduction a une variable pour chaque fil du circuit.

La formule ϕ produite par l'algorithme de réduction est le ET entre la variable de sortie du circuit et la conjonction des clauses décrivant l'opération de chaque porte. Pour le circuit de la figure, la formule est

$$\begin{aligned} \phi = & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \\ & \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4) \\ & \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) . \end{aligned}$$

Étant donné un circuit C , il est immédiat de produire une telle formule ϕ en temps polynomial.

Pourquoi le circuit C est-il satisfaisable exactement quand la formule ϕ est satisfaisable ? Si C possède une assignation satisfaisante, chaque fil du circuit a une valeur bien définie et la sortie du circuit est 1. Donc, l'assignation de valeurs de fil aux variables de ϕ fait que chaque clause de ϕ est évaluée à 1, et donc que leur conjonction est aussi évaluée à 1. Réciproquement, s'il existe une assignation qui donne une évaluation 1 pour ϕ , le circuit C est satisfaisable de par une démonstration analogue. On a donc montré que CIRCUIT-SAT \leq_P SAT, ce qui termine la démonstration. \square

b) Satisfaisabilité 3-CNF

On peut démontrer la NP-complétude de nombreux problèmes par réduction à partir de la satisfaisabilité de formule. L'algorithme de réduction doit cependant gérer toute formule donnée en entrée, ce qui peut conduire à considérer un nombre de cas très important. Il est donc souvent souhaitable de réduire à partir d'un langage restreint de formules booléennes, de façon à considérer moins de cas. Bien sûr, il ne faut pas restreindre le langage au point de le rendre résoluble en temps polynomial. Un langage commode est celui de la satisfaisabilité 3-CNF, ou 3-CNF-SAT.

On définit la satisfaisabilité 3-CNF à l'aide des termes suivants. Un *littéral* dans une formule booléenne est une variable ou sa négation. Une formule booléenne se trouve dans sa *forme normale conjonctive*, ou *forme CNF*, si elle est exprimée comme un ET de clauses, chacune d'elles étant le OU d'un ou plusieurs littéraux. Une formule booléenne se trouve dans sa *forme normale conjonctive d'ordre 3*, ou *3-CNF*, si chaque clause comporte exactement trois littéraux distincts.

Par exemple, la formule booléenne

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

est en forme 3-CNF. La première de ses trois clauses est $(x_1 \vee \neg x_1 \vee \neg x_2)$, qui contient les trois littéraux x_1 , $\neg x_1$ et $\neg x_2$.

Dans 3-CNF-SAT, on cherche à savoir si une formule booléenne ϕ donnée en forme 3-CNF est satisfaisable. Le théorème suivant montre qu'un algorithme polynomial qui peut déterminer la satisfaisabilité d'une formule booléenne a très peu de chances d'exister, même quand elle est exprimée dans cette forme normale simple.

Théorème 34.10 *Le problème de la satisfaisabilité des formules booléennes sous forme 3-CNF est NP-complet.*

Démonstration : La démonstration utilisée au théorème 34.9 pour montrer que $\text{SAT} \in \text{NP}$ s'applique aussi pour montrer que $3\text{-CNF-SAT} \in \text{NP}$. Donc, d'après le lemme 34.8, il suffit de montrer que $\text{SAT} \leq_p 3\text{-CNF-SAT}$.

L'algorithme de réduction se divise en trois grandes parties. Chaque étape transforme progressivement la formule donnée en entrée ϕ en la forme normale conjonctive d'ordre 3.

La première étape est similaire à celle utilisée pour démontrer $\text{CIRCUIT-SAT} \leq_p \text{SAT}$ au théorème 34.9. On commence par construire un arbre binaire « d'analyse » pour la formule ϕ , en plaçant les littéraux sur les feuilles et les connecteurs sur les nœuds internes. La figure 34.11 montre ce type d'arbre d'analyse pour la formule

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \quad (34.3)$$

Si la formule d'entrée contient une clause comme le OU de plusieurs littéraux, on peut utiliser l'associativité pour parenthéser l'expression complètement de sorte que tout nœud interne de l'arbre résultant possède 1 ou 2 enfants. L'arbre binaire d'analyse peut maintenant être vu comme un circuit pour le calcul de la fonction.

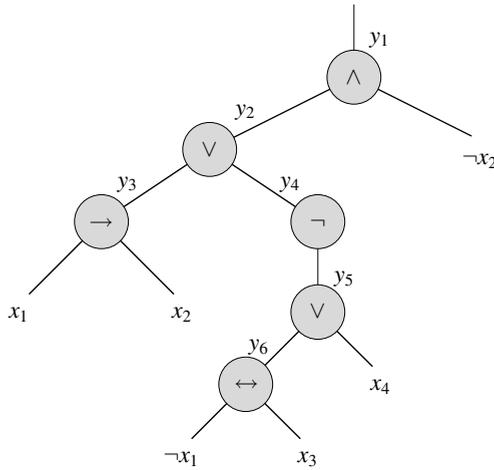


Figure 34.11 L'arbre correspondant à la formule

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 .$$

En imitant la réduction mise en œuvre lors de la démonstration du théorème 34.9, on introduit une variable y_i pour la sortie de chaque nœud interne. Ensuite, on réécrit la formule originale ϕ comme le ET entre la variable racine et une conjonction de clauses décrivant l'action de chaque nœud. Pour la formule (34.3), l'expression qui en résulte est

$$\begin{aligned} \phi' = & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) . \end{aligned}$$

Observez que la formule ϕ' ainsi obtenue est une conjonction de clauses ϕ'_i , possédant chacune au plus 3 littéraux. La seule contrainte supplémentaire est que chaque clause soit un OU de littéraux.

La deuxième étape de la réduction convertit chaque clause ϕ'_i dans sa forme normale conjonctive. On construit une table de vérité pour ϕ'_i en évaluant toutes les assignations possibles à ses variables. Chaque ligne de la table de vérité représente une assignation possible des variables de la clause, plus la valeur de la clause pour cette assignation. En utilisant les éléments de la table de vérité qui sont évalués à 0, on construit une formule en **forme normale disjonctive** (ou **DNF**), c'est à dire un OU de ET, qui est équivalente à $\neg\phi'_i$. Cette formule est ensuite convertie en une formule CNF ϕ''_i , en faisant appel aux lois de Morgan (B.2) pour complémenter tous les littéraux et changer les OU en ET et les ET en OU.

Dans notre exemple, on convertit la clause $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ en forme CNF de la façon suivante. La table de vérité pour ϕ'_1 est donnée à la figure 34.12. La formule DNF équivalente à $\neg\phi'_1$ est

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2).$$

En appliquant les lois de DeMorgan, on obtient la formule CNF

$$\begin{aligned} \phi''_1 = & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2), \end{aligned}$$

qui est équivalente à la clause initiale ϕ'_1 .

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figure 34.12 La table de vérité de la clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

Chaque clause ϕ'_i de la formule ϕ' est maintenant convertie en une formule CNF ϕ''_i , et ϕ' est donc équivalente à la formule CNF ϕ'' composée de la conjonction des ϕ''_i . Par ailleurs, chaque clause de ϕ'' possède au plus 3 littéraux.

La troisième et dernière étape de la réduction transforme encore la formule, de façon que chaque clause contienne *exactement* 3 littéraux distincts. La formule 3-CNF finale ϕ''' est construite à partir des clauses de la formule CNF ϕ'' . Elle utilise également deux variables auxiliaires que nous appellerons p et q . Pour chaque clause C_i de ϕ'' , on inclut les clauses suivantes dans ϕ''' :

- Si C_i a 3 littéraux distincts, alors on se contente d'inclure C_i comme clause de ϕ''' .
- Si C_i a 2 littéraux distincts, autrement dit si $C_i = (l_1 \vee l_2)$, où l_1 et l_2 sont des littéraux, on inclut dans ϕ''' les clauses $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$. Les littéraux p et $\neg p$ servent simplement à respecter la contrainte syntaxique qui veut qu'il y ait exactement 3 littéraux distincts par clause : $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ est équivalent à $(l_1 \vee l_2)$, aussi bien pour $p = 0$ que pour $p = 1$.
- Si C_i ne possède qu'1 seul littéral l , alors on inclut dans ϕ''' les clauses $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$. Notez que, quelles que soient les valeurs de p et q , la conjonction de ces quatre clauses donne une évaluation de l .

En se penchant sur chacune de ces trois étapes, on peut voir que la formule 3-CNF ϕ''' est satisfaisable si et seulement si ϕ est satisfaisable. Comme pour la réduction de CIRCUIT-SAT à SAT, la construction de ϕ' à partir de ϕ dans la première étape conserve la satisfaisabilité. La deuxième étape produit une formule CNF ϕ'' qui est algébriquement équivalente à ϕ' . La troisième étape produit une formule 3-CNF ϕ''' qui

est effectivement équivalente à ϕ'' , puisqu'une assignation quelconque des variables p et q produit une formule qui est algébriquement équivalente à ϕ'' .

On doit également montrer que la réduction peut être calculée en temps polynomial. La construction de ϕ' à partir de ϕ introduit au plus 1 variable et 1 clause par connecteur de ϕ . La construction de ϕ'' à partir de ϕ' peut introduire au plus 8 clauses dans ϕ'' pour chaque clause de ϕ' , puisque chaque clause de ϕ' a au plus 3 variables et que la table de vérité de chaque clause contient au plus $2^3 = 8$ lignes. La construction de ϕ''' à partir de ϕ'' introduit au plus 4 clauses dans ϕ''' pour chaque clause de ϕ'' . Donc, la taille de la formule résultante ϕ''' est polynomiale par rapport à la longueur de la formule d'origine. Chacune des constructions peut être facilement accomplie en temps polynomial. \square

Exercices

34.4.1 On considère la réduction directe (en temps non polynomial) dans la démonstration du théorème 34.9. Décrire un circuit de taille n qui, quand on le convertit en une formule par cette méthode, génère une formule dont la taille est exponentielle en n .

34.4.2 Montrer la formule 3-CNF qui résulte de l'application de la méthode du théorème 34.10 sur la formule (34.3).

34.4.3 Le professeur Jagger se propose de montrer $\text{SAT} \leq_P \text{3-CNF-SAT}$ en ne faisant appel qu'à la technique de la table de vérité utilisée dans la démonstration du théorème 34.10, et pas aux autres étapes. Autrement dit, le professeur propose de prendre la formule booléenne ϕ , de former une table de vérité pour ses variables, de calculer à partir de la table de vérité une formule 3-CNF équivalente à $\neg\phi$, puis de prendre la négation et d'appliquer les lois de DeMorgan pour produire une formule 3-CNF équivalente à ϕ . Montrer que cette stratégie ne donne pas une réduction à temps polynomial.

34.4.4 Montrer que le problème consistant à déterminer si une formule booléenne est une tautologie est complet pour co-NP. (*conseil* : Voir exercice 34.3.7.)

34.4.5 Montrer que le problème consistant à déterminer la satisfaisabilité des formules booléennes en forme normale disjonctive est résoluble en temps polynomial.

34.4.6 Supposez que quelqu'un découvre un algorithme à temps polynomial pour décider de la satisfaisabilité des formules. Décrire comment utiliser cet algorithme pour trouver des assignations satisfaisantes en temps polynomial.

34.4.7 Soit 2-CNF-SAT l'ensemble des formules booléennes satisfaisables exprimées sous forme CNF avec exactement 2 littéraux par clause. Montrer que $2\text{-CNF-SAT} \in \text{P}$. Rendre votre algorithme aussi efficace que possible. (*conseil* : Observer que $x \vee y$ est équivalent à $\neg x \rightarrow y$. Réduire 2-CNF-SAT à un problème de graphe orienté qui soit efficacement résoluble.)

34.5 PROBLÈMES NP-COMPLETS

Les problèmes NP-complets apparaissent dans des domaines variés : logique booléenne, graphes, arithmétique, conception de réseau, ensembles et partitions, stockage et accès, séquençement et planification, programmation mathématique, algèbre et théorie des nombres, jeux et puzzles, automates et théorie des langages, optimisation des programmes, biologie, chimie, physique, etc. Dans cette section, nous utiliserons la méthodologie de la réduction pour fournir des preuves de la NP-complétude de divers problèmes concernant les graphes et le partitionnement d'ensemble.

La figure 34.13 esquisse la structure des preuves de NP-complétude traitées dans cette section et dans la section 34.4. On démontre que chaque langage de la figure est NP-complet via réduction à partir du langage qui pointe vers lui. A la racine, on trouve CIRCUIT-SAT, dont nous avons démontré qu'il est NP-complet au théorème 34.7.

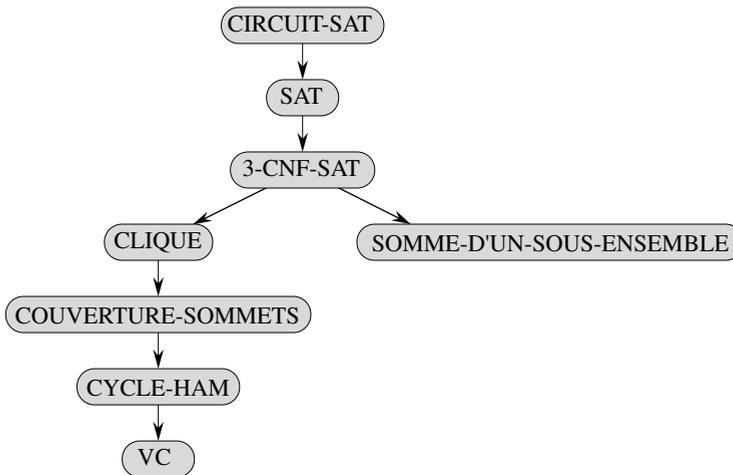


Figure 34.13 La structure des preuves de NP-complétude des sections 34.4 et 34.5. Toutes les démonstrations se déduisent en fin de compte de la réduction depuis la NP-complétude de CIRCUIT-SAT.

34.5.1 Problème de la clique

Une *clique* dans un graphe non orienté $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ de sommets, dont chaque paire est reliée par une arête de A . Autrement dit, une clique est un sous-graphe complet de G . La *taille* d'une clique est le nombre de sommets qu'elle contient. Le *problème de la clique* est le problème d'optimisation consistant à trouver une clique de taille maximale dans un graphe. En tant que problème de décision, on veut simplement savoir si une clique de taille k donnée existe dans le

graphe. La définition formelle est

$$\text{CLIQUE} = \{ \langle G, k \rangle : G \text{ est un graphe contenant une clique de taille } k \} .$$

Un algorithme naïf pour déterminer si un graphe $G = (S, A)$ à $|S|$ sommets contient une clique de taille k consiste à énumérer tous les k -sous-ensembles de S et regarder pour chacun s'il forme une clique. Le temps d'exécution de cet algorithme est $\Omega(k^2 \binom{|S|}{k})$, ce qui est polynomial si k est une constante. Cependant, dans le cas général, k pourrait être proche de $|S|$, auquel cas l'algorithme s'exécute en temps supra-polynomial. Comme on pourrait s'y attendre, il y a très peu de chances pour qu'un algorithme efficace existe pour résoudre le problème de la clique.

Théorème 34.11 *Le problème de la clique est NP-complet.*

Démonstration : Pour montrer que $\text{CLIQUE} \in \text{NP}$, pour un graphe donné $G = (S, A)$, on utilise l'ensemble $S' \subseteq S$ des sommets de la clique comme certificat pour G . Vérifier que S' est une clique peut être accompli en temps polynomial, en testant si pour toute paire $u, v \in S'$ l'arête (u, v) appartient à A .

Nous montrons ensuite que $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$, ce qui montre que le problème de la clique est NP-difficile. Sachant le peu de rapport apparent entre les formules logiques et les graphes, le fait que nous soyons capables de démontrer ce résultat paraît quelque peu surprenant.

L'algorithme de réduction commence avec une instance de 3-CNF-SAT. Soit $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$ une formule booléenne sous forme 3-CNF à k clauses. Pour $r = 1, 2, \dots, k$, chaque clause C_r possède exactement trois littéraux distincts l'_1, l'_2 et l'_3 . Nous allons construire un graphe G tel que ϕ soit satisfaisable si et seulement si G possède une clique de taille k .

Le graphe $G = (S, A)$ est construit de la manière suivante. Pour chaque clause $C_r = (l'_1 \vee l'_2 \vee l'_3)$ de ϕ , on place un triplet de sommets v'_1, v'_2 et v'_3 dans S . On place une arête entre deux sommets v'_i et v'_j si les deux conditions suivantes sont satisfaites :

- v'_i et v'_j sont dans des triplets différents, autrement dit si $r \neq s$, et
- leur littéraux correspondants sont *cohérents*, autrement dit l'_i n'est pas la négation de l'_j .

Ce graphe peut se calculer facilement à partir de ϕ en temps polynomial. A titre d'exemple de cette construction, si l'on se donne

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) ,$$

alors G est le graphe représenté à la figure 34.14.

Il faut montrer que ce passage de ϕ à G est une réduction. D'abord, supposons que ϕ ait une assignation satisfaisante. Alors, chaque clause C_r contient au moins un littéral l'_i ayant la valeur 1, et chaque littéral de ce type correspond à un sommet v'_i . En prenant l'un de ces littéraux ayant la valeur vrai dans chaque clause, on obtient un ensemble S' de k sommets. Nous affirmons que S' est une clique. Pour deux sommets quelconques $v'_i, v'_j \in S'$, où $r \neq s$, l'assignation satisfaisante associe la valeur 1 aux deux littéraux l'_i et l'_j correspondants, et les littéraux ne peuvent donc pas être complémentaires. Donc, d'après la construction de G , l'arête (v'_i, v'_j) appartient à A .

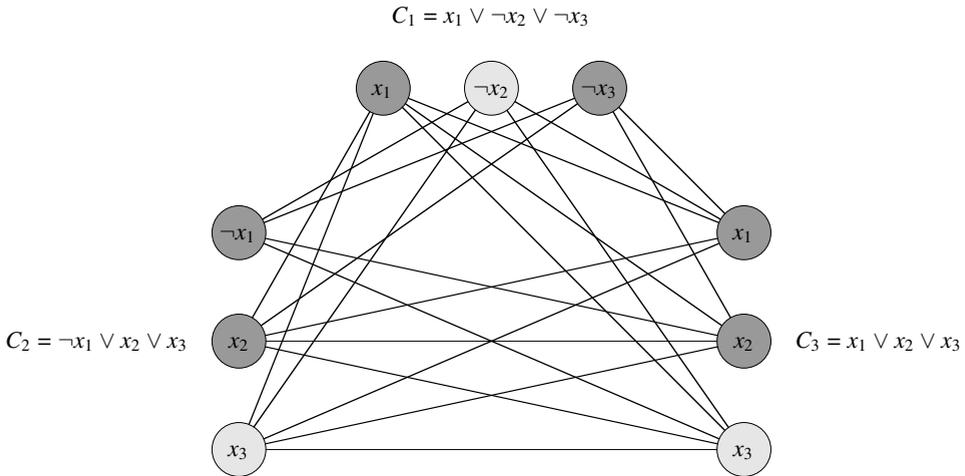


Figure 34.14 Le graphe G issu de la formule 3-CNF $\phi = C_1 \wedge C_2 \wedge C_3$, où $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ et $C_3 = (x_1 \vee x_2 \vee x_3)$, lors de la réduction de 3-CNF-SAT à CLIQUE. L'assignation $\langle x_1 = 0 \text{ ou } 1, x_2 = 0, x_3 = 1 \rangle$ est satisfaisante. Cette assignation vérifie C_1 avec $\neg x_2$ et vérifie C_2 et C_3 avec x_3 , ce qui correspond à la clique dont les sommets sont en gris clair.

Réciproquement, supposons que G possède une clique S' de taille k . Aucune arête de G ne relie des sommets appartenant au même triplet, et donc S' contient exactement un sommet par triplet. On peut affecter 1 à chaque littéral l'_i tel que $v'_i \in S'$ sans craindre d'affecter 1 à la fois à un littéral et à son complément, puisque G ne contient aucune arête entre deux littéraux incohérents. Chaque clause est satisfaite, et ϕ est donc satisfaite. (Toute variable ne correspondant à aucun sommet de la clique peut être initialisée à une valeur arbitraire.) \square

Dans l'exemple de la figure 34.14, une assignation satisfaisante de ϕ a $x_2 = 0$ et $x_3 = 1$. Une clique correspondante de taille $k = 3$ est constituée des sommets correspondant au $\neg x_2$ de la première clause, au x_3 de la deuxième clause et au x_3 de la troisième clause. Comme la clique ne contient pas de sommets correspondant à x_1 ou à $\neg x_1$, on peut mettre x_1 à 0 ou à 1 dans cette assignation satisfaisante. Observez que, dans la preuve du théorème 34.11, on a réduit une instance arbitraire de 3-CNF-SAT à une instance de CLIQUE ayant une structure particulière. On pourrait penser que l'on a montré que CLIQUE est NP-difficile uniquement pour les graphes où les sommets vont par trois et où il n'y a pas d'arêtes entre deux sommets d'un même triplet. En fait, on a montré que CLIQUE est NP-difficile uniquement pour ce cas particulier, mais cette preuve suffit pour montrer que CLIQUE est NP-difficile pour des graphes quelconques. Pourquoi? Si l'on avait un algorithme à temps polynomial qui résolve CLIQUE sur des graphes généraux, il résoudrait aussi CLIQUE sur des graphes spéciaux.

En revanche, cela n'aurait pas été suffisant que de réduire des instances de 3-CNF-SAT ayant une structure spéciale à des instances générales de CLIQUE. Pourquoi ? On aurait pu avoir le cas où les instances de 3-CNF-SAT à partir desquelles on aurait choisi de réduire soient « faciles », auquel cas on n'aurait pas réduit un problème NP-difficile à CLIQUE.

Observez aussi que la réduction a utilisé l'instance de 3-CNF-SAT mais pas la solution. C'eût été une erreur, pour la réduction à temps polynomial, que d'être basée sur le fait de savoir si la formule ϕ est satisfaisable, vu que nous ne savons pas comment déterminer cette information en temps polynomial.

34.5.2 Problème de la couverture de sommets

Une *couverture de sommets* d'un graphe non orienté $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ tel que si $(u, v) \in A$, alors $u \in S'$ ou $v \in S'$ (ou les deux). Autrement dit, chaque sommet « couvre » ses arêtes incidentes et une couverture de sommets pour G est un ensemble de sommets qui couvre toute les arêtes de A . La *taille* d'une couverture de sommets est le nombre de sommets qui la composent. Par exemple, le graphe de la figure 34.15(b) a une couverture de sommets $\{w, z\}$ de taille 2.

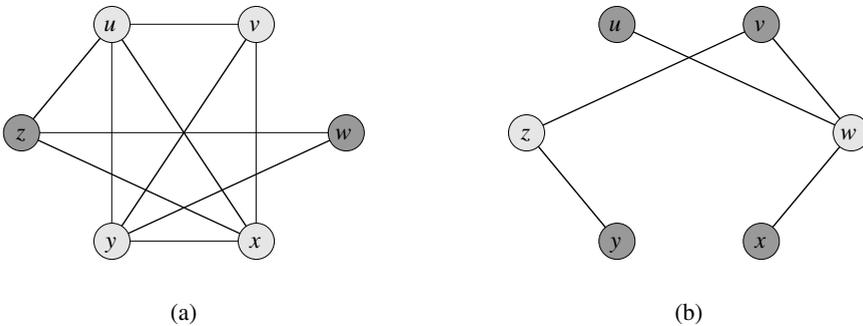


Figure 34.15 Réduction de CLIQUE à COUVERTURE-SOMMETS. **(a)** Un graphe non orienté $G = (S, A)$ avec une clique $S' = \{u, v, x, y\}$. **(b)** Le graphe \bar{G} produit par l'algorithme de réduction ayant pour couverture de sommets $S - S' = \{w, z\}$.

Le *problème de la couverture de sommets* consiste à trouver une couverture de sommet de taille minimale dans un graphe donné. Si l'on énonce ce problème d'optimisation comme un problème de décision, on souhaite déterminer si un graphe possède une couverture de sommets d'une taille k donnée. Si on lui donne l'aspect formel d'un langage, on aura

$$\text{COUVERTURE-SOMMETS} = \{ \langle G, k \rangle : \text{le graphe } G \text{ possède une couverture de sommets de taille } k \} .$$

Le théorème suivant établit que ce problème est NP-complet.

Théorème 34.12 *Le problème de la couverture de sommets est NP-complet.*

Démonstration : On commence par montrer COUVERTURE-SOMMETS \in NP. Supposons qu'on ait un graphe $G = (S, A)$ et un entier k . Le certificat que nous choisissons est la couverture de sommets $S' \subseteq S$ elle-même. L'algorithme de vérification affirme que $|S'| = k$, puis il vérifie, pour chaque arête $(u, v) \in A$, que $u \in S'$ ou $v \in S'$. Cette vérification peut être effectuée en temps polynomial sans astuce particulière.

On démontre que le problème de la couverture de sommets est NP-difficile en montrant que CLIQUE \leq_p COUVERTURE-SOMMETS. Cette réduction est fondée sur la notion de graphe complémentaire. Étant donné un graphe non orienté $G = (S, A)$, on définit le **complémentaire** de G comme $\overline{G} = (S, \overline{A})$ où $\overline{A} = \{(u, v) : u, v \in S, u \neq v, \text{ et } (u, v) \notin A\}$. Autrement dit, \overline{G} est le graphe contenant exactement les arêtes qui ne sont pas dans G . La figure 34.15 montre un graphe et son complémentaire, et illustre la réduction de CLIQUE à COUVERTURE-SOMMETS.

L'algorithme de réduction prend en entrée une instance $\langle G, k \rangle$ du problème de la clique. Il calcule le complémentaire \overline{G} , ce qui se fait facilement en temps polynomial. Le résultat de l'algorithme de réduction est l'instance $\langle \overline{G}, |S| - k \rangle$ du problème de la couverture de sommets. Pour compléter la démonstration, on montre que cette transformation est bien une réduction : le graphe G possède une clique de taille k si et seulement si le graphe \overline{G} possède une couverture de sommets de taille $|S| - k$.

Supposons que G ait une clique $S' \subseteq S$ avec $|S'| = k$. Nous affirmons que $S - S'$ est une couverture de sommets de \overline{G} . Soit (u, v) une arête quelconque de \overline{A} . Alors, $(u, v) \notin A$, ce qui implique qu'au moins un des sommets u et v n'appartient pas à S' , puisque toute paire de sommets de S' est reliée par une arête de A . De manière équivalente, l'un au moins des sommets u ou v se trouve dans $S - S'$, ce qui signifie que l'arête (u, v) est couverte par $S - S'$. Comme (u, v) a été choisi arbitrairement dans \overline{A} , toute arête de \overline{A} est couverte par un sommet de $S - S'$. Donc, l'ensemble $S - S'$, qui a une taille $|S| - k$, forme une couverture de sommets pour \overline{G} .

Réciproquement, supposons que \overline{G} ait une couverture de sommets $S' \subseteq S$, où $|S'| = |S| - k$. Alors, quels que soient $u, v \in S$, si $(u, v) \in \overline{A}$, on a $u \in S'$ ou $v \in S'$ ou les deux. La contraposée de cette implication est : quels que soient $u, v \in S$, si $u \notin S'$ et $v \notin S'$, alors $(u, v) \in A$. Autrement dit, $S - S'$ est une clique et a une taille $|S| - |S'| = k$. \square

Puisque COUVERTURE-SOMMETS est NP-complet, on ne s'attend pas à trouver un algorithme polynomial permettant de déterminer une couverture de sommets de taille minimale. Toutefois, la section 35.1 présentera un « algorithme approché » polynomial, qui produit des solutions « approchées » pour le problème de la couverture de sommets. La taille d'une couverture de sommets produite par l'algorithme est au plus deux fois la taille minimale d'une couverture de sommets.

Il ne faut donc pas perdre espoir sous prétexte qu'un problème est NP-complet. Il peut exister un algorithme approché polynomial capable de trouver des solutions quasi optimales, même si la recherche d'une solution optimale est un problème NP-complet. Le chapitre 35 donnera plusieurs algorithmes approchés pour les problèmes NP-difficiles.

34.5.3 Problème du cycle hamiltonien

On revient à présent au problème du cycle hamiltonien défini à la section 34.2.

Théorème 34.13 *Le problème du cycle hamiltonien est NP-complet.*

Démonstration : On commence par montrer que CYCLE-HAM appartient à NP. Étant donné un graphe $G = (S, A)$, notre certificat sera la séquence des $|S|$ sommets qui constituent le cycle hamiltonien. L’algorithme de vérification teste si cette séquence contient chaque sommet de S exactement une fois et si, quand on répète le premier sommet à la fin, elle forme un cycle dans G . Autrement dit, il teste s’il y a une arête entre chaque paire de sommets consécutifs, ainsi qu’entre les premier et dernier sommets. Cette validation peut s’effectuer en temps polynomial.

Montrons maintenant que COUVERTURE-SOMMETS \leq_P CYCLE-HAM, ce qui prouvera que CYCLE-HAM est NP-complet. Étant donné un graphe non orienté $G = (S, A)$ et un entier k , on construit un graphe non orienté $G' = (S', A')$ qui a un cycle hamiltonien si et seulement si G a une couverture de sommets de taille k .

Notre construction est basée sur un *ga*, qui est une partie d’un graphe dotée de certaines propriétés.

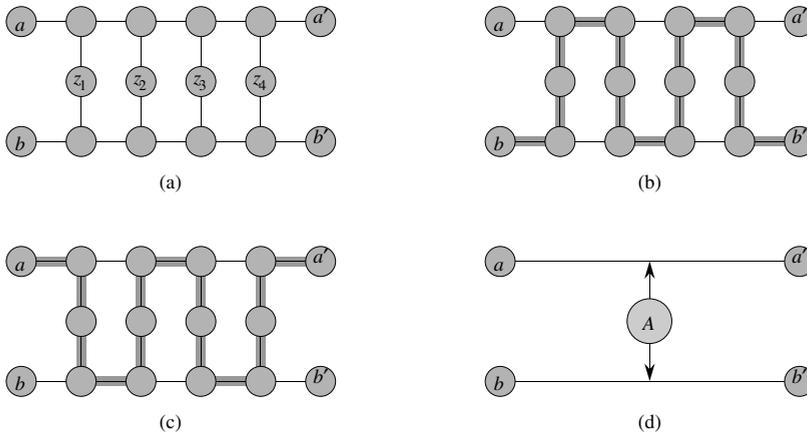


Figure 34.16 (a) Le gadget utilisé pour la réduction du problème de la couverture de sommets au problème du cycle hamiltonien. Une arête $(u, v) \in G$ correspond au gadget W_{uv} du graphe G' créé dans la réduction. (a) Le gadget, avec les divers sommets étiquetés. (b)–(d) Les chemins ombrés sont les seuls chemins possibles à travers le gadget qui incluent tous les sommets, en supposant que les seules connexions du gadget vers le reste de G' passent par les sommets $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ et $[v, u, 6]$.

La figure 34.16(a) montre le gadget que nous employons. Pour chaque arête $(u, v) \in A$, le graphe G' que nous construisons contient un exemplaire de ce gadget, que nous représentons par W_{uv} . Notons chaque sommet de W_{uv} par $[u, v, i]$ ou $[v, u, i]$, avec $1 \leq i \leq 6$, de sorte que chaque gadget W_{uv} contient 12 sommets. Le gadget W_{uv} contient aussi les 14 arêtes montrées à la figure 34.16(a).

En plus de la structure interne du gadget, définissons les propriétés qui nous intéressent en limitant les connexions entre le gadget et le reste du graphe G' que nous construisons. En particulier, seuls les sommets $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ et $[v, u, 6]$ auront des arêtes incidentes provenant de l'extérieur de W_{uv} . Tout cycle hamiltonien de G' devra traverser les arcs de W_{uv} de l'une des trois façons montrées sur les figures 34.16(b)–(d). Si le cycle entre par le sommet $[u, v, 1]$, il doit sortir par le sommet $[u, v, 6]$ et, soit il visite les 12 sommets du gadget (figure 34.16(b)), soit il visite les six sommets $[u, v, 1]$ à $[u, v, 6]$ (figure 34.16(c)). Dans ce dernier cas, le cycle devra rentrer dans le gadget pour visiter les sommets $[v, u, 1]$ à $[v, u, 6]$. De même, si le cycle entre par le sommet $[v, u, 1]$, il doit sortir par le sommet $[v, u, 6]$ et, soit il visite les 12 sommets du gadget (figure 34.16(d)), soit il visite les six sommets $[v, u, 1]$ à $[v, u, 6]$ (figure 34.16(c)). Il n'existe pas d'autre chemin à travers le gadget qui visite les 12 sommets. En particulier, il est impossible de construire deux chemins sans sommet commun, dont l'un relie $[u, v, 1]$ à $[v, u, 6]$ et l'autre relie $[v, u, 1]$ à $[u, v, 6]$, tels que l'union des deux chemins contienne tous les sommets du gadget.

Les seuls autres sommets de S' autres que ceux des gadgets sont des *sommets sélecteur* s_1, s_2, \dots, s_k . On utilise les arêtes incidentes aux sommets sélecteur de G' pour sélectionner les k sommets de la couverture de G .

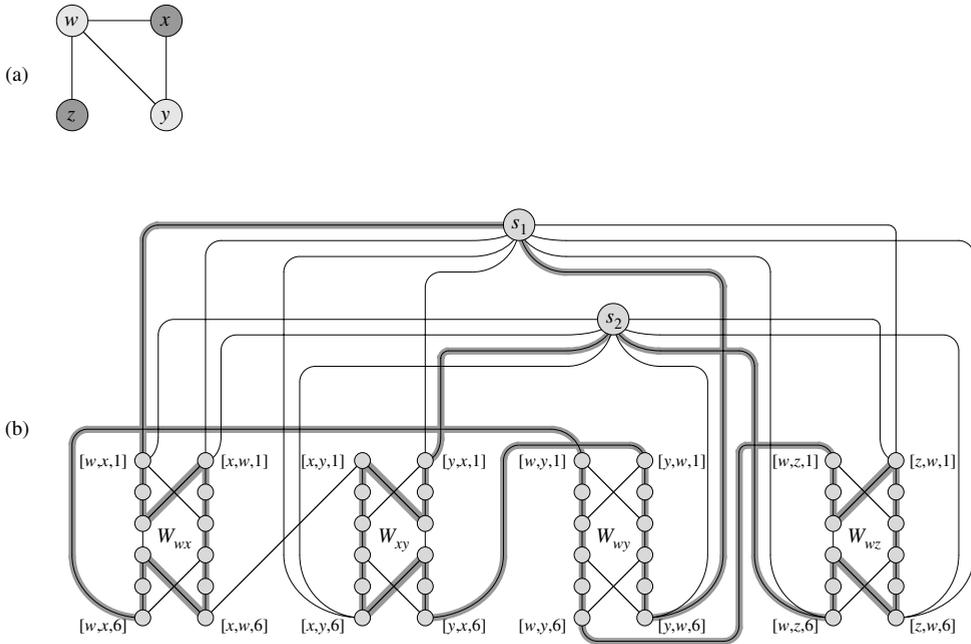


Figure 34.17 La réduction d'une instance du problème de la couverture de sommets à une instance du problème du cycle hamiltonien. (a) Un graphe non orienté G avec une couverture de sommets de taille 2, composée des sommets en gris clair w et y . (b) Le graphe non orienté G' produit par la réduction, avec le chemin hamiltonien correspondant à la couverture de sommets en ombré. La couverture $\{w, y\}$ correspond aux arêtes $(s_1, [w, x, 1])$ et $(s_2, [y, x, 1])$ qui apparaissent dans le cycle hamiltonien.

Outre les arêtes des gadgets, il y a deux autres types d'arête dans A' , montrés à la figure 34.17. Primo, pour chaque sommet $u \in S$, on ajoute des arêtes pour joindre des paires de gadgets afin de former un chemin qui contienne tous les gadgets correspondant à des arêtes incidentes à u dans G . On ordonne arbitrairement les sommets adjacents à chaque sommet $u \in S$ sous la forme $u^{(1)}, u^{(2)}, \dots, u^{(\text{degré}(u))}$, où $\text{degré}(u)$ est le nombre de sommets adjacents à u . On crée un chemin dans G' qui passe par tous les gadgets correspondant à des arêtes incidentes à u en ajoutant à A' les arêtes $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \leq i \leq \text{degré}(u) - 1\}$. Sur la figure 34.17, par exemple, on ordonne les sommets adjacents à w sous la forme x, y, z , et donc le graphe G' de la partie (b) de la figure contient les arêtes $([w, x, 6], [w, y, 1])$ et $([w, y, 6], [w, z, 1])$. Pour chaque sommet $u \in S$, ces arêtes de G' forment un chemin qui contient tous les gadgets correspondant à des arêtes incidentes à u dans G .

L'idée sous-jacente à ces arêtes est que, si on choisit un sommet $u \in S$ dans la couverture de sommets de G , alors on peut construire un chemin de $[u, u^{(1)}, 1]$ à $[u, u^{(\text{degré}(u))}, 6]$ dans G' qui « couvre » tous les gadgets correspondant à des arcs incidents à u . En d'autres termes, pour chacun de ces gadgets, par exemple $W_{u, u^{(i)}}$, le chemin soit inclut les 12 sommets (si u est dans la couverture, mais pas $u^{(i)}$), soit inclut uniquement les six sommets $[u, u^{(i)}, 1], [u, u^{(i)}, 2], \dots, [u, u^{(i)}, 6]$ (si u et $u^{(i)}$ appartiennent tous les deux à la couverture).

Le dernier type d'arête dans A' relie le premier sommet $[u, u^{(1)}, 1]$ et le dernier $[u, u^{(\text{degré}(u))}, 6]$ de chacun de ces chemins à chacun des sommets sélecteur. Autrement dit, on inclut les arêtes

$$\begin{aligned} & \{(s_j, [u, u^{(1)}, 1]) : u \in S \text{ et } 1 \leq j \leq k\} \\ & \cup \{(s_j, [u, u^{(\text{degré}(u))}, 6]) : u \in S \text{ et } 1 \leq j \leq k\}. \end{aligned}$$

Ensuite, on montre que la taille de G' est polynomiale par rapport à celle de G , et donc que l'on peut construire G' en temps polynomial par rapport à la taille de G . Les sommets de G' sont ceux des gadgets, plus les sommets sélecteur. Chaque gadget contient 12 sommets, et il y a $k \leq |S|$ sommets sélecteur, soit un total de

$$\begin{aligned} |S'| &= 12 |A| + k \\ &\leq 12 |A| + |S| \end{aligned}$$

sommets. Les arêtes de G' sont ceux des gadgets, ceux qui relient les gadgets, plus ceux qui relient les sommets sélecteur aux gadgets. Il y a 14 arêtes dans chaque gadget, soit $14 |A|$ tous gadgets confondus. Pour chaque sommet $u \in S$, il y a $\text{degré}(u) - 1$ arêtes entre des gadgets ; donc, si l'on somme sur tous les sommets de S , il y a

$$\sum_{u \in S} (\text{degré}(u) - 1) = 2 |A| - |S|$$

arêtes entre des gadgets. Enfin, il y a deux arêtes pour chaque paire composée d'un sommet sélecteur et d'un sommet de S , soit $2k |S|$ de tels sommets. Le nombre total d'arêtes de G' est donc

$$\begin{aligned} |A'| &= (14 |A|) + (2 |A| - |S|) + (2k |S|) \\ &= 16 |A| + (2k - 1) |S| \\ &\leq 16 |A| + (2 |S| - 1) |S|. \end{aligned}$$

Montrons maintenant que la transformation de G en G' est une réduction. Autrement dit, il faut montrer que G a une couverture de sommets de taille k si et seulement si G' a un cycle hamiltonien.

Supposons que $G = (S, A)$ ait une couverture de sommets $S^* \subseteq S$ de taille k .

Soit $S^* = \{u_1, u_2, \dots, u_k\}$. Comme le montre la figure 34.17, on forme un cycle hamiltonien dans G en incluant les arêtes suivantes ⁽⁸⁾ pour chaque sommet $u_j \in S^*$. Inclure les arêtes $\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \leq i \leq \text{degré}(u_j)\}$, qui relie tous les gadgets correspondant à des arêtes incidentes à u_j . Inclure également les arêtes situées dans ces gadgets, comme le montrent les figures 34.16(b)–(d), selon que l'arête est couverte par un ou deux sommets de S^* . Le cycle hamiltonien contient aussi les arêtes

$$\begin{aligned} & \{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \leq j \leq k\} \\ & \cup \{(s_{j+1}, [u_j, u_j^{(\text{degré}(u_j))}, 6]) : 1 \leq j \leq k-1\} \\ & \cup \{(s_1, [u_k, u_k^{(\text{degré}(u_k))}, 6])\}. \end{aligned}$$

En inspectant la figure 34.17, le lecteur pourra vérifier que ces arêtes forment un cycle. Le cycle commence en s_1 , visite tous les gadgets correspondant à des arêtes incidentes à u_1 , puis visite s_2 , visite tous les gadgets correspondant à des arêtes incidentes à u_2 , etc. jusqu'à ce qu'il revienne sur s_1 . Chaque gadget visité est visité une ou deux fois, selon qu'il y a un ou deux sommets de S^* qui couvrent son arête correspondant. Comme S^* est une couverture de sommets pour G , chaque arête de A est incident à un certain sommet de S^* , et donc le cycle visite chaque sommet de chaque gadget de G' . Comme le cycle visite aussi chaque sommet sélecteur, il est hamiltonien.

Inversement, supposons que $G' = (S', A')$ ait un cycle hamiltonien $C \subseteq A'$. Nous affirmons que l'ensemble

$$V^* = \{u \in S : (s_j, [u, u^{(1)}, 1]) \in C \text{ pour un certain } 1 \leq j \leq k\} \quad (34.4)$$

est une couverture de sommets pour G . Pour le voir, partitionnons C en chemins maximaux qui partent d'un certain sommet sélecteur s_i , traversent une arête $(s_i, [u, u^{(1)}, 1])$ pour un certain $u \in S$, puis finissent en un sommet sélecteur s_j sans passer par un autre sommet sélecteur. Appelons un tel chemin un « chemin couverture ». Selon la façon dont est construit G' , chaque chemin couverture doit commencer en un certain s_i , emprunter l'arête $(s_i, [u, u^{(1)}, 1])$ pour un certain sommet $u \in S$, passer par tous les gadgets correspondant à des arêtes de A incidentes à u , puis finir en un certain sommet sélecteur s_j . Désignons ce chemin couverture par p_u et, en vertu de l'équation (34.4), plaçons u dans S^* . Chaque gadget visité par p_u doit être W_{uv} ou W_{vu} pour un certain $v \in S$. Pour chaque gadget visité par p_u , ses sommets sont visités par un ou deux chemins couverture. S'ils sont visités par un seul chemin couverture, alors l'arête $(u, v) \in A$ est couverte dans G par le sommet u . Si deux chemins couverture visitent le gadget, alors l'autre chemin couverture doit être p_v , ce qui implique que $v \in S^*$ et que l'arête $(u, v) \in A$ est couverte par u et par v . Comme chaque sommet de chaque gadget est visité par un certain chemin couverture, on voit que chaque arête de A est couverte par un certain sommet de S^* . \square

(8) Techniquement parlant, on définit un cycle en termes de sommets et non d'arêtes (voir section B.4). À des fins de clarté, on fait ici un abus de langage et on définit le cycle hamiltonien en termes d'arêtes.

34.5.4 Problème du voyageur de commerce

Dans le *problème du voyageur de commerce*, qui est très proche du problème du cycle hamiltonien, un représentant doit visiter n villes. En modélisant le problème par un graphe complet à n sommets, on peut dire que le représentant souhaite faire une *tournée*, ou cycle hamiltonien, en visitant chaque ville exactement une fois, et en terminant sa tournée dans la ville de départ. Le voyage entre la ville i et la ville j a un coût entier $c(i, j)$, et le représentant souhaite effectuer la tournée dont le coût total est minimal, le coût total étant la somme des coûts individuels le long de chaque arête de la tournée. Par exemple, dans la figure 34.18, une tournée de coût minimal sera $\langle u, w, v, x, u \rangle$, avec un coût total égal à 7. Le langage formel pour le problème de décision correspondant est

$$\text{VC} = \{ \langle G, c, k \rangle : G = (S, A) \text{ est un graphe complet,} \\ c \text{ est une fonction de } S \times S \rightarrow \mathbf{Z}, \\ k \in \mathbf{Z}, \text{ et} \\ G \text{ a une tournée de coût au plus égal à } k \} .$$

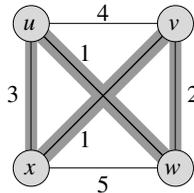


Figure 34.18 Une instance du problème du voyageur de commerce. Les arêtes ombrées représentent une tournée de coût minimal, égal à 7.

Le théorème suivant montre qu'un algorithme rapide pour le problème du voyageur de commerce a très peu de chances d'exister.

Théorème 34.14 *Le problème du voyageur de commerce est NP-complet.*

Démonstration : On commence par montrer que VC appartient à NP. Étant donnée une instance du problème, on utilise comme certificat la séquence des n sommets de la tournée. L'algorithme de vérification teste si cette séquence contient chaque sommet exactement une fois, cumule les coûts des arêtes, puis vérifie que la somme vaut au plus k . Ce processus peut sans problème être mis en œuvre en temps polynomial.

Pour démontrer que VC est NP-difficile, on montre que CYCLE-HAM \leq_P VC. Soit $G = (S, A)$ une instance de CYCLE-HAM. On construit une instance de VC comme suit. On forme le graphe complet $G' = (S, A')$, où $A' = \{(i, j) : i, j \in S \text{ et } i \neq j\}$, et on définit la fonction de coût c par

$$c(i, j) = \begin{cases} 0 & \text{si } (i, j) \in A, \\ 1 & \text{si } (i, j) \notin A. \end{cases}$$

(Notez que, G étant non orienté, il n'a pas de boucles et donc que $c(v, v) = 1$ pour tous les sommets $v \in S$.) L'instance de VC est alors $(G', c, 0)$, qui peut être facilement formée en temps polynomial.

Montrons à présent que le graphe G contient un cycle hamiltonien si et seulement si le graphe G' possède une tournée de coût au plus égale à 0. Supposons que le graphe G possède un cycle hamiltonien h . Chaque arête de h appartient à A et a donc un coût nul dans G' . Donc, h est une tournée de G' de coût nul. Réciproquement, supposons que le graphe G' possède une tournée h' de coût au plus égal à 0. Puisque les coûts des arêtes de A' valent 0 et 1, le coût de la tournée h' vaut exactement 0 et chaque arête de la tournée doit avoir un coût 0. Donc, h' contient uniquement des arêtes de A . On en conclut que h est un cycle hamiltonien du graphe G . \square

34.5.5 Problème de la somme de sous-ensemble

Notre prochain problème NP-complet sera de nature arithmétique. Dans le **problème de la somme de sous-ensemble**, on se donne un ensemble fini $S \subset \mathbf{N}$ et un **objectif** $t \in \mathbf{N}$. On veut savoir s'il existe un sous-ensemble $S' \subseteq S$ dont la somme des éléments est t .

Par exemple, si $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ et $t = 138457$, alors le sous-ensemble $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ est une solution.

Comme d'habitude, on définit le problème en tant que langage :

SOMME-SOUS-ENSEMBLE =

$$\{ \langle S, t \rangle : \text{il existe un sous-ensemble } S' \subseteq S \text{ tel que } t = \sum_{s \in S'} s \} .$$

Comme pour tout problème arithmétique, il est important de se rappeler que notre encodage standard suppose que les entiers en entrée sont codés en binaire. En ayant cela à l'esprit, nous pouvons montrer que le problème de la somme de sous-ensemble a peu de chances d'avoir un algorithme rapide.

Théorème 34.15 *Le problème de la somme de sous-ensemble est NP-complet.*

Démonstration : Pour montrer que SOMME-SOUS-ENSEMBLE est dans NP, pour une instance $\langle S, t \rangle$ du problème, nous prenons le sous-ensemble S' comme certificat. Vérifier que $t = \sum_{s \in S'} s$ peut se faire à l'aide d'un algorithme de vérification en temps polynomial.

Montrons maintenant que $3\text{-CNF-SAT} \leq_P \text{SOMME-SOUS-ENSEMBLE}$. Étant donnée une formule 3-CNF ϕ sur les variables x_1, x_2, \dots, x_n avec les clauses C_1, C_2, \dots, C_k , chacune contenant exactement trois littéraux distincts, l'algorithme de réduction construit une instance $\langle S, t \rangle$ du problème de la somme de sous-ensemble telle que ϕ est satisfaisable si et seulement si il y a un sous-ensemble de S dont la somme fait exactement t . Sans nuire à la généralité, on va faire deux hypothèses simplificatrices pour la formule ϕ . Primo, aucune clause ne contient à la fois une variable et sa négation, car une telle clause est automatiquement vérifiée par toute assignation de valeurs aux variables. Secundo, chaque variable apparaît dans au moins une clause, car sinon peu importe la valeur qui lui est assignée.

La réduction crée deux nombres dans l'ensemble S pour chaque variable x_i et deux nombres dans S pour chaque clause C_j . Nous créerons des nombres en base 10, chaque nombre contenant ici $n + k$ chiffres et chaque chiffre correspondant soit à une variable soit à une clause. La base 10 (et d'autres bases, comme nous le verrons) a la propriété dont nous avons besoin pour empêcher les retenues de passer des chiffres de poids faible aux chiffres de poids fort.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Figure 34.19 Réduction de 3-CNF-SAT à SOMME-SOUS-ENSEMBLE. La formule 3-CNF est $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, avec $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ et $C_4 = (x_1 \vee x_2 \vee x_3)$. Une assignation satisfaisante de ϕ est $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. L'ensemble S produit par la réduction se compose des nombres en base 10 affichés sur la figure; en lisant du haut vers le bas, $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. L'objectif t est 1114444. Le sous-ensemble $S' \subseteq S$, en gris clair, contient v'_1, v'_2 et v_3 , correspondant à l'assignation satisfaisante. Il contient aussi les variables d'écart $s_1, s'_1, s'_2, s_3, s_4$ et s'_4 pour atteindre la valeur cible 4 dans les chiffres étiquetés C_1 à C_4 .

Ainsi que le montre la figure 34.19, on construit l'ensemble S et l'objectif t de la façon suivante. On étiquette chaque position numérique par une variable ou une clause. Les k chiffres les moins significatifs sont étiquetés par les clauses, et les n chiffres les plus significatifs sont étiquetés par les variables.

- L'objectif t a un 1 dans chaque chiffre étiqueté par une variable, et un 4 dans chaque chiffre étiqueté par une clause.
- Pour chaque variable x_i , il y a deux entiers, v_i et v'_i , dans S . Chacun a un 1 dans le chiffre étiqueté par x_i et des 0 dans les autres chiffres de la variable. Si le littéral x_i apparaît dans la clause C_j , alors le chiffre étiqueté par C_j dans v_i contient un 1. Si le

littéral $\neg x_i$ apparaît dans la clause C_j , alors le chiffre étiqueté par C_j dans v'_i contient un 1. Tous les autres chiffres étiquetés par des clauses dans v_i et v'_i sont 0.

Toutes les valeurs v_i et v'_i dans l'ensemble S sont uniques. Pourquoi ? Pour $l \neq i$, aucune valeur v_l ou v'_l ne peuvent être égales à v_i et v'_i dans les n chiffres les plus significatifs. En outre, en raison de nos hypothèses simplificatrices précédentes, aucun v_i et v'_i ne peut être égal dans tous les k chiffres les moins significatifs. Si v_i et v'_i étaient égaux, alors x_i et $\neg x_i$ devraient apparaître dans exactement le même ensemble de clauses. Mais on suppose qu'aucune clause ne contient à la fois x_i et $\neg x_i$ et que x_i ou $\neg x_i$ apparaît dans une certaine clause ; Donc, il doit y avoir une certaine clause C_j pour laquelle v_i et v'_i diffèrent.

- Pour chaque clause C_j , il y a deux entiers, s_j et s'_j dans S . Chacun a des 0 dans tous les chiffres autres que celui étiqueté par C_j . Pour s_j , il y a un 1 dans le chiffre C_j et s'_j a un 2 dans ce chiffre. Ces entiers sont des « variables d'écart », que l'on utilise pour obtenir chaque position numérique étiquetée par une clause à ajouter à la valeur cible de 4.

Un simple examen de la figure 34.19 montre que toutes les valeurs s_j et s'_j de S sont uniques dans l'ensemble S .

Notez que la plus grande somme de chiffres dans une quelconque position numérique est 6, ce qui se produit dans les chiffres étiquetés par des clauses (trois 1 provenant des valeurs v_i et v'_i , plus 1 et 2 provenant des valeurs s_j et s'_j). Comme l'on interprète ces nombres en base 10, il ne peut pas y avoir de retenues qui passent des chiffres de poids faible vers les chiffres de poids fort. ⁽⁹⁾

La réduction peut se faire en temps polynomial. L'ensemble S contient $2n + 2k$ valeurs, dont chacune a $n + k$ chiffres, et le temps pour produire chaque chiffre est polynomial en $n + k$. L'objectif t a $n + k$ chiffres, et la réduction produit chacun en temps constant.

Montrons maintenant que la formule 3-CNF ϕ est satisfaisable si et seulement si il y a un sous-ensemble $S' \subseteq S$ dont la somme est t . Primo, supposons que ϕ ait une assignation satisfaisante. Pour $i = 1, 2, \dots, n$, si $x_i = 1$ dans cette assignation, alors on inclut v_i dans S' . Sinon, on inclut v'_i . Autrement dit, on inclut dans S' très précisément les valeurs v_i et v'_i qui correspondent à des littéraux ayant la valeur 1 dans l'assignation satisfaisante. Ayant inclus v_i ou v'_i , mais pas les deux, pour tout i , et ayant mis 0 dans les chiffres étiquetés par des variables dans tous les s_j et s'_j , on voit que, pour chaque chiffre étiqueté par une variable, la somme des valeurs de S' doit être 1, ce qui concorde avec ces chiffres dans l'objectif t . Comme chaque clause est satisfaite, il y a un certain littéral de la clause qui a la valeur 1. Donc, chaque chiffre étiqueté par une clause a au moins un 1 qui a été ajouté à sa somme par une valeur v_i ou v'_i de S' . En fait, il peut y avoir 1, 2 ou 3 littéraux valant 1 dans chaque clause, et donc chaque chiffre étiqueté par une clause a une somme de 1, 2 ou 3 provenant des valeurs v_i et v'_i de S' . (Sur la figure 34.19, par exemple, les littéraux $\neg x_1$, $\neg x_2$ et x_3 ont la valeur 1 dans une assignation satisfaisante.) Chacune des clauses C_1 et C_4 contient un tel littéral et un seul, et donc v'_1 , v'_2 et v_3 contribuent globalement pour 1 à la somme dans les chiffres pour C_1 et C_4 . La clause C_2 contient deux tels littéraux, et donc v'_1 , v'_2 et v_3 contribuent pour 2 à la somme dans le chiffre pour C_2 . La clause C_3 contient trois tels littéraux, et donc v'_1 , v'_2 et v_3 contribuent pour 3 à la somme dans le chiffre pour C_3 .) On atteint

(9) En fait, toute base b , avec $b \geq 7$, ferait l'affaire. L'instance au début de cette sous-section est l'ensemble S et l'objectif t de la figure 34.19 interprété en base 7, avec S affiché dans l'ordre trié.

l'objectif 4 dans chaque chiffre étiqueté par la clause C_j en incluant dans S' le sous-ensemble non vide idoine de variables d'écart $\{s_j, s'_j\}$. (Sur la figure 34.19, S' inclut $s_1, s'_1, s'_2, s_3, s_4$ et s'_4 .) Comme nous avons trouvé des correspondances pour l'objectif sur tous les chiffres de la somme, et qu'il ne peut pas y avoir de retenue, les valeurs de S' ont pour somme t .

Supposons maintenant qu'il y ait un sous-ensemble $S' \subseteq S$ dont la somme fasse t . S' doit inclure une seule des deux valeurs v_i et v'_i pour tout $i = 1, 2, \dots, n$; autrement, la somme des chiffres étiquetés par des variables ne ferait pas 1. Si $v_i \in S'$, faisons $x_i = 1$. Sinon, $v'_i \in S'$ et l'on fait $x_i = 0$. Nous affirmons que chaque clause C_j , pour $j = 1, 2, \dots, k$, est vérifiée par cette assignation. Pour prouver cette affirmation, notons que, pour atteindre une somme de 4 dans le chiffre étiqueté par C_j , le sous-ensemble S' doit inclure au moins une valeur v_i ou v'_i qui a un 1 dans le chiffre étiqueté par C_j ; en effet, les contributions cumulées des variables d'écart s_j et s'_j donnent un total au plus égal à 3. Si S' inclut un v_i qui a un 1 à cette position, alors le littéral x_i apparaît dans la clause C_j . Comme on a fait $x_i = 1$ quand $v_i \in S'$, la clause C_j est satisfaite. Si S' inclut un v'_i qui a un 1 dans cette position, alors le littéral $\neg x_i$ apparaît dans C_j . Comme on a fait $x_i = 0$ quand $v'_i \in S'$, ici aussi la clause C_j est satisfaite. Donc, toutes les clauses de ϕ sont satisfaites, ce qui termine la démonstration. \square

Exercices

34.5.1 Le *problème du sous-graphe isomorphe* prend deux graphes G_1 et G_2 et demande si G_1 est isomorphe à un sous-graphe de G_2 . Montrer que le problème du sous-graphe isomorphe est NP-complet.

34.5.2 Étant donné une matrice $m \times n$ à valeurs entières A et un m -vecteur à coefficients entiers b , le *problème de la programmation entière 0-1* demande s'il existe un n -vecteur x dont les éléments sont dans $\{0, 1\}$ tel que $Ax \leq b$. Démontrer que la programmation entière 0-1 est un problème NP-complet. (*conseil* : Effectuer la réduction à partir de 3-CNF-SAT.)

34.5.3 Le *problème de la programmation linéaire entière* est le même que celui de la programmation entière 0-1 donné à l'exercice 34.5.2, sauf que maintenant les valeurs du vecteur x sont des entiers quelconques. En suposant que le problème de la programmation entière 0-1 soit NP-difficile, montrer que le problème de la programmation linéaire entière est NP-complet.

34.5.4 Montrer que le problème de la somme de sous-ensemble est résoluble en temps polynomial si la valeur cible t est exprimée en unaire.

34.5.5 Le *problème du partitionnement d'ensemble* prend en entrée un ensemble E de nombres. La question est de savoir si les nombres peuvent être partitionnés en deux ensembles A et $\bar{A} = E - A$ tels que $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$. Montrer que le problème du partitionnement d'ensemble est NP-complet.

34.5.6 Montrer que le problème du chemin hamiltonien est NP-complet.

34.5.7 Le *problème du plus long cycle élémentaire* consiste à déterminer un cycle élémentaire (pas de sommets répétés) de longueur maximale dans un graphe. Montrer que ce problème est NP-complet.

34.5.8 Dans le problème de la *semi satisfaisabilité 3-CNF*, on a une formule 3-CNF ϕ à n variables et m clauses, avec m pair. On veut savoir s'il existe une assignation de vérité pour les variables de ϕ telle que la moitié exactement des clauses aient une évaluation de 0, et l'autre moitié une évaluation de 1. Prouver que le problème de la semi satisfaisabilité 3-CNF est NP-complet.

PROBLÈMES

34.1. Ensemble stable

Un *ensemble stable* d'un graphe $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ de sommets tels que chaque arête de A soit incidente à au plus un sommet de S' . Le *problème du stable maximum* consiste à trouver un ensemble stable de taille maximale dans G .

- a. Formuler un problème de décision correspondant au problème du stable maximum, et démontrer qu'il est NP-complet. (*Conseil* : Réduire à partir du problème de la clique.)
- b. Supposons qu'on se donne un sous-programme du genre « boîte noire » capable de résoudre le problème de décision défini à la partie (a). Donner un algorithme capable de trouver un ensemble stable de taille maximale. Le temps d'exécution de votre algorithme devra être polynomial en $|S|$ et $|A|$, les requêtes à la boîte noire étant comptabilisées comme une seule étape.

Bien que le problème de décision du stable maximum soit NP-complet, certains cas spéciaux sont résolubles en temps polynomial.

- c. Donner un algorithme efficace permettant de résoudre le problème du stable maximum quand chaque sommet de G a un degré 2. Analyser le temps d'exécution et démontrer que votre algorithme est correct.
- d. Donner un algorithme efficace pour résoudre le problème du stable maximum quand G est biparti. Analyser le temps d'exécution et démontrer que l'algorithme est correct. (*Conseil* : Faire appel aux résultats de la section 26.3.)

34.2. Bonnie and Clyde

Bonnie et Clyde viennent de piller une banque. Ils ont un sac plein d'argent et veulent le partager. Pour chacun des scénarios suivants, donner un algorithme à temps polynomial, ou alors prouver que le problème est NP-complet. L'entrée de chaque scénario est la liste des n éléments du sac, avec la valeur de chacun.

- a. Il y a n pièces, mais de deux types seulement : des pièces de x dollars et des pièces de y dollars. Les deux complices veulent se partager l'argent équitablement.
- b. Il y a n pièces, la valeur de chacune étant une puissance entière positive de 2 ; les pièces possibles sont donc des pièces de 1 dollar, 2 dollars, 4 dollars, etc. Les deux complices veulent se partager l'argent équitablement.
- c. Il y a n chèques qui, ô miracle, sont tous libellés à l'ordre de « Bonnie ou Clyde ». Les deux complices veulent se répartir les chèques de façon à avoir la même somme chacun.
- d. Il y a n chèques, comme à la partie (c), mais cette fois les deux complices sont prêts à accepter un partage inégal, à condition que la différence entre les deux parts ne fasse pas plus de 100 dollars.

34.3. Coloration d'un graphe

Un k -coloration d'un graphe non orienté $G = (S, A)$ est une fonction $c : S \rightarrow \{1, 2, \dots, k\}$ telle que $c(u) \neq c(v)$ pour toute arête $(u, v) \in A$. Autrement dit, les nombres $1, 2, \dots, k$ représentent les k couleurs, et des sommets adjacents doivent avoir des couleurs différentes. Le **problème de la coloration d'un graphe** consiste à déterminer le nombre minimal de couleurs nécessaires pour colorier un graphe donné.

- a. Donner un algorithme efficace pour déterminer une 2-coloration d'un graphe si elle existe.
- b. Transformer le problème de coloration d'un graphe en un problème de décision. Montrer que le problème de décision peut être résolu en temps polynomial si et seulement si on peut résoudre le problème de coloration d'un graphe en temps polynomial.
- c. Soit 3-COULEURS le langage des graphes qui peuvent être 3-coloriés. Montrer que, si 3-COULEURS est NP-complet, alors le problème de décision de la partie (b) est NP-complet.

Pour démontrer que 3-COULEURS est NP-complet, on utilise une réduction depuis 3-CNF-SAT. Étant donnée une formule ϕ de m clauses et n variables x_1, x_2, \dots, x_n , on construit un graphe $G = (S, A)$ de la manière suivante. L'ensemble S est composé d'un sommet pour chaque variable, un sommet pour la négation de chaque variable, 5 sommets pour chaque clause, et 3 sommets spéciaux : VRAI, FAUX et ROUGE. Les arêtes du graphe sont de deux types : les arêtes « littéral » qui sont indépendantes des clauses, et les arêtes « clause » qui dépendent des clauses. Les arêtes littérales forment un triangle sur les sommets spéciaux et forment aussi un triangle sur $x_i, \neg x_i$ et ROUGE pour $i = 1, 2, \dots, n$.

- d. Montrer que, dans toute 3-coloriation c d'un graphe contenant les arêtes littérales, une seule des variables et sa négation sont coloriées respectivement $c(\text{VRAI})$ et $c(\text{FAUX})$ (ou vice versa). Montrer que, pour toute assignation de vérité pour ϕ , il existe une 3-coloration du graphe ne contenant que les arêtes littérales.

Le gadget de la figure 34.20 sert à appliquer la condition correspondant à une clause $(x \vee y \vee z)$. Chaque clause demande une copie unique des 5 sommets dessinés en foncé sur la figure ; comme le montre la figure, ils sont reliés aux littéraux de la clause et au sommet spécial VRAI.

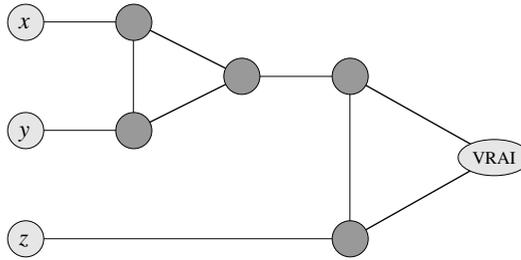


Figure 34.20 Le gadget du problème 34.3, correspondant à une clause $(x \vee y \vee z)$.

- e. Montrer que, si chacun des x , y et z est colorié $c(\text{VRAI})$ ou $c(\text{FAUX})$, alors le gadget est 3-coloriable si et seulement si au moins l'un des x , y et z est colorié $c(\text{VRAI})$.
- f. Finir de démontrer que 3-COULEURS est NP-complet.

34.4. Ordonnement avec profit et délais limite

Supposez que vous ayez une machine et un ensemble de n tâches a_1, a_2, \dots, a_n . Chaque tâche a_j a une durée d'exécution t_j , un profit p_j et une date d'échéance d_j . La machine ne peut traiter qu'une seule tâche à la fois, et a_j doit s'exécuter sans interruption pendant t_j unités de temps consécutives. Si vous achevez a_j avant l'échéance d_j , vous recevez un profit p_j ; vous ne recevez rien si vous terminez après l'échéance. Le problème d'optimisation est le suivant : connaissant les durées d'exécution, les profits et les dates d'échéance d'un ensemble de n tâches, on veut trouver un ordonnancement qui permette d'exécuter toutes les tâches avec un profit maximal.

- a. Énoncer ce problème sous la forme d'un problème de décision.
- b. Montrer que le problème de décision est NP-complet.
- c. Donner un algorithme à temps polynomial pour le problème de décision, en supposant que toutes les durées d'exécution sont des entiers entre 1 et n . (*Conseil* : Utiliser la programmation dynamique.)
- d. Donner un algorithme à temps polynomial pour le problème d'optimisation, en supposant que toutes les durées d'exécution sont des entiers entre 1 et n .

NOTES

Le livre de Garey et Johnson [110] est un magnifique guide pour la NP-complétude, étudiant la théorie en détail et donnant une liste de problèmes connus en 1979 pour être NP-complets.

La démonstration du théorème 34.13 s'inspire de ce livre et la liste des domaines de problèmes NP-complets, donnée au début de la section 34.5, est tirée de leur table des matières. Johnson a écrit une suite de 23 articles dans la revue *Journal of Algorithms* entre 1981 et 1992, qui font état des derniers progrès en matière de NP-complétude. Hopcroft, Motwani et Ullman [153], Lewis et Papadimitriou [204], Papadimitiou [236], et Sipser [279] contiennent de bonnes études de la NP-complétude dans le contexte de la théorie de la complexité. Aho, Hopcroft et Ullman [5] couvrent également la NP-complétude et donnent plusieurs réductions, notamment une réduction pour le problème de la couverture de sommets à partir du problème du cycle hamiltonien.

La classe P fut introduite en 1964 par Cobham [64] et, indépendamment, par Edmonds [84] en 1965, qui introduisit également la classe NP et conjectura que $P \neq NP$. La notion de NP-complétude fut proposée en 1971 par Cook [67], qui donna les premières démonstrations de NP-complétude pour la satisfaisabilité de formule et la satisfaisabilité 3-CNF. Cette notion fut découverte indépendamment par Levin [203], qui donna une preuve de la NP-complétude d'un problème de pavage. Karp [173] introduisit la méthodologie des réductions en 1972 et démontra la grande variété des problèmes NP-complets. L'article de Karp incluait les premières démonstrations de la NP-complétude des problèmes de la clique, de la couverture de sommets et du cycle hamiltonien. Depuis lors, de nombreux chercheurs ont démontré la NP-complétude de centaines de problèmes. Dans un discours prononcé à l'occasion du sixième anniversaire de Karp en 1995, Papadimitriou dit : « Chaque année, près de 6000 articles contiennent le terme « NP-complet » dans leur titre, leur résumé, leur thésaurus, etc. Aucun des termes « compilateur », « base de données », « expert », « réseau neuronal » ou « système d'exploitation » ne peut se targuer d'une telle célébrité. »

Des travaux récents en matière de théorie de la complexité ont rénové la notion de complexité du calcul de solutions approchées. Ces travaux donnent une nouvelle définition de NP à l'aide de « démonstrations vérifiables probabilistiquement ». Avec cette nouvelle définition, pour des problèmes comme celui de la clique, de la couverture de sommets, du voyageur de commerce avec inégalité triangulaire, etc., le calcul de bonnes solutions approchées est NP-difficile et, partant, pas plus facile que le calcul de solutions optimales. On trouvera une introduction à ce domaine dans la thèse d'Arora [19], dans un chapitre d'Arora et Lund [149], dans une étude d'Arora [20], dans un livre publié par Mayr, Promel et Steger [214], ainsi que dans une étude de Johnson [167].

Chapitre 35

Algorithmes d'approximation

De nombreux problèmes d'intérêt pratique, bien que NP-difficiles, sont trop importants pour qu'on les abandonne pour la seule raison qu'il n'existe pas d'algorithme efficace pour obtenir une solution optimale. Si un problème est NP-difficile, il y a peu de chances de trouver un algorithme à temps polynomial capable de le résoudre exactement, mais cela n'implique pas que tout espoir soit perdu. Il existe au moins deux approches pour contourner la NP-difficulté. Primo, si les entrées réelles sont petites, un algorithme à temps d'exécution exponentiel peut parfaitement convenir. Secundo, il se peut que l'on puisse trouver des solutions *quasi optimales* à temps polynomial (dans le cas le plus défavorable, ou en moyenne). En pratique, on peut souvent se contenter d'une solution quasi optimale. Un algorithme qui retourne des solutions quasi optimales est appelé **algorithme d'approximation**. Ce chapitre présente des algorithmes d'approximation à temps polynomial pour plusieurs problèmes NP-difficiles.

a) Garanties de performance pour algorithmes d'approximation

Supposons qu'on travaille sur un problème d'optimisation dans lequel chaque solution potentielle a un coût positif, et qu'on souhaite trouver une solution quasi optimale. Selon le problème, une solution optimale pourra être définie comme étant soit celle de coût maximal, soit celle de coût minimal ; le problème pourra donc être un problème de maximisation ou de minimisation.

On dit qu'un algorithme d'approximation a une **garantie de performance** $\rho(n)$ si, pour toute entrée de taille n , le coût C de la solution produite par l'algorithme est éloigné d'un facteur de $\rho(n)$ du coût C^* d'une solution optimale :

$$\max \left(\frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n) . \quad (35.1)$$

Un algorithme qui atteint une garantie de performance $\rho(n)$ est dit **algorithme d'approximation $\rho(n)$** . Ces définitions de garantie de performance et d'algorithme d'approximation s'appliquent aux problèmes de minimisation comme aux problèmes de maximisation. Pour un problème de maximisation, $0 < C \leq C^*$, et le ratio C^*/C donne le facteur par lequel le coût d'une solution optimale est supérieur à celui de la solution approchée. De même, pour un problème de minimisation, $0 < C^* \leq C$ et le ratio C/C^* donne le facteur par lequel le coût de la solution approchée est supérieur à celui de la solution optimale. Comme on suppose que toutes les solutions ont un coût positif, ces ratios sont toujours bien définis. La garantie de performance d'un algorithme d'approximation n'est jamais inférieure à 1, puisque $C/C^* < 1$ implique $C^*/C > 1$. Un algorithme d'approximation 1 donne donc une solution optimale, et un algorithme d'approximation ayant un gros ratio risque de produire une solution rien moins qu'optimale.

Pour de nombreux problèmes, l'on a inventé des algorithmes d'approximation à temps polynomial dotés de garanties de performance petites et constantes ; pour d'autres problèmes, en revanche, les meilleurs algorithmes d'approximation à temps polynomial que l'on connaisse ont des garanties qui croissent en fonction de la taille n de l'entrée. Un exemple de ce genre de problème est la couverture d'ensemble, qui sera traitée à la section 35.3.

Pour certains problèmes NP-difficiles, l'on peut trouver des algorithmes d'approximation à temps polynomial pouvant atteindre des garanties de plus en plus petites en consommant de plus en plus de temps de calcul. Autrement dit, il y a un compromis entre le temps de calcul et la qualité de l'approximation. Un exemple en est donné par le problème de la somme de sous-ensembles, traité à la section 35.5. Cette situation est suffisamment importante pour qu'elle ait un nom à elle.

Un **schéma d'approximation** pour un problème d'optimisation est un algorithme d'approximation qui prend en entrée non seulement une instance du problème, mais aussi une valeur $\varepsilon > 0$ telle que, pour tout ε fixé à l'avance, le schéma est un algorithme d'approximation $(1 + \varepsilon)$. On dit qu'un schéma d'approximation est un **schéma d'approximation polynomial** si, pour tout $\varepsilon > 0$ fixé, le schéma s'exécute en temps polynomial par rapport à la taille n de l'entrée.

Le temps d'exécution d'un schéma d'approximation polynomial peut croître très rapidement quand ε décroît. Par exemple, le temps d'exécution d'un schéma d'approximation polynomial pourrait être $O(n^{2/\varepsilon})$. Dans l'idéal, si ε décroît d'un facteur constant, le temps d'exécution permettant d'atteindre l'approximation désirée ne devrait pas croître de plus d'un facteur constant. Autrement dit, on aimerait que le temps d'exécution soit polynomial en $1/\varepsilon$, et pas seulement en n .

On dit qu'un schéma d'approximation est un **schéma d'approximation entièrement polynomial** si le temps d'exécution est polynomial à la fois en $1/\varepsilon$ et en n , où n est la taille de l'entrée. Par exemple, le schéma pourrait avoir un temps d'exécution de $(1/\varepsilon)^2 n^3$. Avec un tel schéma, toute diminution de ε d'un facteur constant peut être atteinte moyennant une augmentation du temps d'exécution d'un facteur constant.

b) Résumé du chapitre

Les quatre premières sections de ce chapitre présenteront quelques exemples d'algorithmes d'approximation à temps polynomial pour des problèmes NP-difficiles, et la cinquième section présentera un schéma d'approximation entièrement polynomial. La section 35.1 commence par une étude du problème de la couverture des sommets, problème NP-difficile qui possède un algorithme d'approximation ayant un ratio de 2. La section 35.2 présente un algorithme d'approximation ayant un ratio de 2 pour le problème du voyageur de commerce, quand la fonction de coût vérifie l'inégalité triangulaire. Elle montre aussi que, sans l'inégalité triangulaire, pour toute constante $\rho \geq 1$ un algorithme d'approximation ρ ne peut pas exister à moins qu'on ait $P = NP$. A la section 35.3, on montrera comment une méthode gloutonne peut servir d'algorithme d'approximation pour le problème de la couverture d'ensemble, donnant une couverture dont le coût le plus défavorable est supérieur d'un facteur logarithmique au coût optimal. La section 35.4 présentera deux autres algorithmes d'approximation. Primo, nous étudierons la version optimisation de la satisfaisabilité 3-CNF et nous donnerons un algorithme randomisé simple qui produit une solution ayant un ratio espéré de $8/7$. Ensuite, nous examinerons une variante pondérée du problème de la couverture de sommets et montrerons comment employer la programmation linéaire pour développer un algorithme d'approximation 2. Enfin, la section 35.5 présentera un schéma d'approximation entièrement polynomial pour le problème de la somme de sous-ensembles.

35.1 PROBLÈME DE LA COUVERTURE DE SOMMETS

Nous avons défini le problème de la couverture de sommets et démontré qu'il était NP-complet à la section 34.5.2. Une **couverture de sommets** d'un graphe non orienté $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ tel que, si (u, v) est une arête de G , alors $u \in S'$ ou $v \in S'$ (ou les deux). La taille d'une couverture de sommets est le nombre de sommets qu'elle contient.

Le **problème de la couverture de sommets** consiste à trouver une couverture de sommets de taille minimale dans un graphe non orienté donné. Une telle couverture de sommets s'appelle une **couverture de sommets optimale**. Ce problème est la version optimisation d'un problème de décision NP-complet.

Bien qu'il puisse être difficile de trouver une couverture de sommets optimale dans un graphe G , il n'est pas trop difficile de trouver une couverture de sommets quasi optimale. L'algorithme d'approximation suivant prend en entrée un graphe non orienté G et retourne une couverture de sommets dont on peut être sûr que la taille ne fait pas plus de deux fois la taille d'une couverture de sommets optimale.

COUVERTURE-SOMMET-APPROCHÉE(G)

```

1   $C \leftarrow \emptyset$ 
2   $A' \leftarrow A[G]$ 
3  tant que  $A' \neq \emptyset$ 
4      faire soit  $(u, v)$  une arête arbitraire de  $A'$ 
5           $C \leftarrow C \cup \{u, v\}$ 
6          supprimer de  $A'$  toutes les arêtes incidentes à  $u$  ou à  $v$ 
7  retourner  $C$ 

```

La figure 35.1 illustre l'action de COUVERTURE-SOMMET-APPROCHÉE. La variable C contient la couverture de sommets en cours de construction. La ligne 1 initialise C à l'ensemble vide. La ligne 2 place dans A' une copie de l'ensemble d'arêtes $A[G]$. La boucle des lignes 3–6 choisit au fur et à mesure une arête (u, v) de A' , ajoute ses extrémités u et v à C , puis supprime toutes les arêtes de A' qui sont couvertes par u ou par v . Le temps d'exécution de cet algorithme est $O(S+A)$, si l'on utilise des listes d'adjacences pour représenter A' .

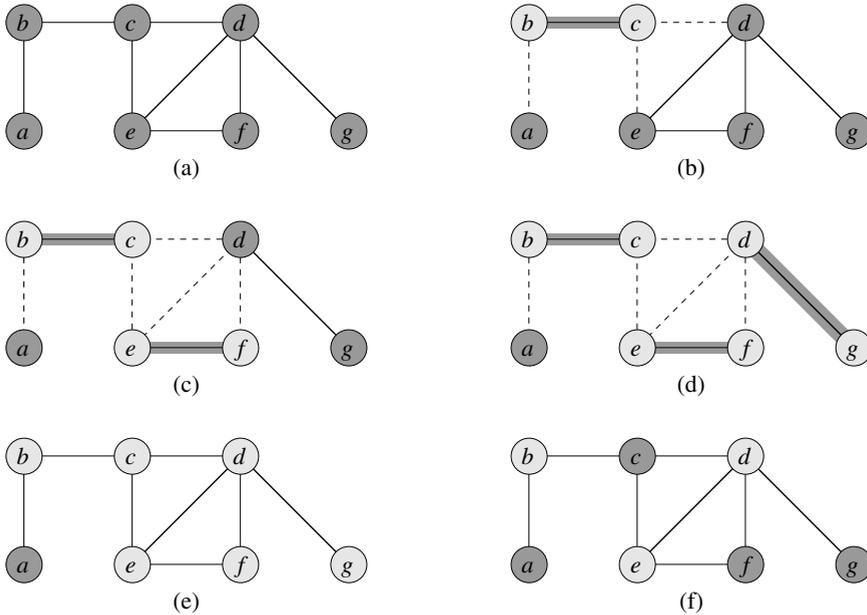


Figure 35.1 L'action de COUVERTURE-SOMMET-APPROCHÉE. (a) Le graphe G en entrée, qui contient 7 sommets et 8 arêtes. (b) L'arête (b, c) , montrée en gras, est la première arête choisie par COUVERTURE-SOMMET-APPROCHÉE. Les sommets b et c , montrés en gris clair, sont ajoutés à l'ensemble C contenant la couverture de sommets en cours de création. Les arêtes (a, b) , (c, e) et (c, d) , montrées en pointillés, sont supprimées car elles ne sont pas couvertes par un sommet de C . (c) L'arête (e, f) est choisie ; les sommets e et f sont ajoutés à C . (d) L'arête (d, g) est choisie ; les sommets d et g sont ajoutés à C . (e) L'ensemble C , qui est la couverture de sommets produite par COUVERTURE-SOMMET-APPROCHÉE, contient les six sommets b, c, d, e, f, g . (f) La couverture de sommets optimale pour ce problème ne contient que trois sommets : b, d et e .

Théorème 35.1 COUVERTURE-SOMMET-APPROCHÉE est un algorithme d'approximation 2 à temps polynomial.

Démonstration : Nous avons déjà montré que l'algorithme tourne en temps polynomial. L'ensemble C de sommets retourné par COUVERTURE-SOMMET-APPROCHÉE est une couverture de sommets, puisque l'algorithme boucle jusqu'à ce que toutes les arêtes de $A[G]$ aient été couvertes par un sommet de C .

Pour voir que COUVERTURE-SOMMET-APPROCHÉE retourne une couverture de sommets qui fait au plus deux fois la taille d'une couverture optimale, soit E l'ensemble des arêtes qui ont été choisies en ligne 4 de COUVERTURE-SOMMET-APPROCHÉE. Pour couvrir les arêtes de E , une couverture de sommets (en particulier, une couverture optimale) doit contenir au moins une extrémité de chaque arête de E . Deux arêtes quelconques de E ne peuvent pas avoir d'extrémité commune, puisqu'une fois qu'une arête est choisie en ligne 4, toutes les autres arêtes qui sont incidentes à ses extrémités sont supprimées de A' en ligne 6. Par conséquent, il n'y a jamais deux arêtes de E qui soient couvertes par le même sommet de C^* , et l'on a le minorant

$$|C^*| \geq |A| \quad (35.2)$$

pour la taille d'une couverture de sommets optimale. Chaque exécution de la ligne 4 choisit une arête telle qu'aucune de ses extrémités ne soit déjà dans C , ce qui donne un majorant (une borne, en fait) de la taille de la couverture de sommets retournée :

$$|C| = 2|A| . \quad (35.3)$$

En combinant les équations (35.2) et (35.3), on obtient

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| , \end{aligned}$$

ce qui démontre le théorème. \square

Arrêtons-nous sur cette démonstration. De prime abord, on pourrait se demander comment il est possible de prouver que la taille de la couverture de sommets retournée par COUVERTURE-SOMMET-APPROCHÉE fasse au plus deux fois la taille d'une couverture optimale, vu que nous ne savons pas ce qu'est la taille de la couverture de sommets optimale. La réponse est que nous utilisons un minorant de la couverture de sommets optimale. Ainsi que l'exercice 35.1.2 vous demandera de le montrer, l'ensemble E des arêtes choisies en ligne 4 de COUVERTURE-SOMMET-APPROCHÉE est en fait un couplage (matching) maximal du graphe G . (Un **couplage maximal** est un couplage qui n'est pas un sous-ensemble propre d'un quelconque autre couplage.) La taille d'un couplage maximal est, comme nous l'avons montré dans la démonstration du théorème 35.1, un minorant de la taille d'une couverture de sommets maximale. L'algorithme retourne une couverture de sommets dont la taille ne fait pas plus de deux fois la taille du couplage maximal E . En établissant une relation entre la taille de la solution retournée et le minorant, nous obtenons notre garantie de performance. Nous réutiliserons cette méthodologie dans des sections ultérieures.

Exercices

35.1.1 Donner un exemple de graphe pour lequel COUVERTURE-SOMMET-APPROCHÉE engendre toujours une solution sous-optimale.

35.1.2 Soit E l'ensemble des arêtes choisies en ligne 4 de COUVERTURE-SOMMET-APPROCHÉE. Prouver que l'ensemble E est un couplage maximal du graphe G .

35.1.3 Le professeur Sourire propose l'heuristique suivante pour résoudre le problème de la couverture de sommets. On choisit de façon répétée un sommet de plus haut degré, et on supprime toutes ses arêtes incidentes. Donner un exemple montrant que l'heuristique du professeur n'a pas une garantie de performance égale à 2. (*Conseil* : : Essayer un graphe bipartite qui a des sommets de degré uniforme à gauche et des sommets de degré variable à droite.)

35.1.4 Donner un algorithme glouton efficace qui trouve en temps linéaire une couverture de sommets optimale pour un arbre.

35.1.5 La démonstration du théorème 34.12 nous a appris que le problème de la couverture de sommets et le problème NP-complet de la clique étaient complémentaires, au sens où une couverture de sommets optimale est le complément d'une clique de taille maximale dans le graphe complémentaire. Cette relation implique-t-elle qu'il existe un algorithme d'approximation à temps polynomial et à ratio d'approximation constant pour le problème de la clique ? Justifiez votre réponse.

35.2 PROBLÈME DU VOYAGEUR DE COMMERCE

Dans le problème du voyageur de commerce, introduit à la section 34.5.4, on se donne un graphe non orienté complet $G = (S, A)$, avec un coût entier positif ou nul $c(u, v)$ associé à chaque arête $(u, v) \in A$, et on doit trouver un cycle hamiltonien (une tournée) dans G qui ait un coût minimal. Pour étendre notre notation, appelons $c(E)$ le coût total des arêtes du sous-ensemble $E \subseteq A$:

$$c(E) = \sum_{(u,v) \in E} c(u, v).$$

Dans nombre de situations pratiques, il est toujours moins coûteux d'aller directement d'un endroit u à un endroit w ; passer par une étape intermédiaire v ne peut pas coûter moins cher. Autrement dit, la suppression d'une étape intermédiaire n'augmente jamais le coût. On formalise cette notion en disant que la fonction coût c vérifie *l'inégalité triangulaire* si, pour trois sommets $u, v, w \in S$ quelconques,

$$c(u, w) \leq c(u, v) + c(v, w).$$

L'inégalité triangulaire est une inégalité naturelle et, dans de nombreuses applications, elle est automatiquement satisfaite. Par exemple, si les sommets du graphe

sont des points du plan et si le coût du trajet entre deux sommets est la distance euclidienne ordinaire entre ceux-ci, alors l'inégalité triangulaire est vérifiée. (Il existe bien des fonctions de coût autres que la distance euclidienne qui vérifient l'inégalité triangulaire.)

Comme le montrera l'exercice 3.5.2, contraindre la fonction de coût à respecter l'inégalité triangulaire n'a pas de conséquence sur le caractère NP-complet du problème du voyageur de commerce. Il existe donc très peu de chances pour que l'on puisse trouver un algorithme à temps polynomial capable de résoudre ce problème exactement. On préférera donc chercher de bons algorithmes d'approximation.

À la section 35.2.1, on examinera un algorithme d'approximation 2 pour le problème du voyageur de commerce avec inégalité triangulaire. À la section 35.2.2, on montrera que, sans l'inégalité triangulaire, un algorithme d'approximation à temps polynomial et à ratio d'approximation constant n'existe pas à moins qu'on ait $P = NP$.

35.2.1 Problème du voyageur de commerce avec inégalité triangulaire

En appliquant la méthodologie de la précédente section, nous allons commencer par calculer une structure (un arbre couvrant minimum) dont le poids est un minorant de la longueur d'une tournée optimale du voyageur de commerce. Nous utiliserons ensuite l'arbre couvrant minimum pour créer une tournée dont le coût ne fait pas plus de deux fois le poids de l'arbre couvrant de poids minimal, pour autant que la fonction de coût satisfasse à l'inégalité triangulaire. L'algorithme suivant met en œuvre cette méthode, appelant l'algorithme d'arbre couvrant de poids minimal ACM-PRIM, vu à la section section 23.2, comme sous-routine.

TOURNÉE-VC-APPROCHÉE(G, c)

- 1 sélectionner un sommet $r \in V[G]$ pour faire office de « racine »
- 2 calculer un ACM T pour G depuis la racine r
en utilisant ACM-PRIM(G, c, r)
- 3 soit L la liste des sommets visités dans un parcours préfixe de T
- 4 **retourner** le cycle hamiltonien H qui visite les sommets dans l'ordre L

Rappelez-vous (section 12.1) que le parcours préfixe d'un arbre visite récursivement tous les sommets, affichant un sommet lors de la première rencontre sans attendre d'avoir visité ses enfants.

La figure 35.2 illustre l'action de TOURNÉE-VC-APPROCHÉE. La partie (a) de la figure montre l'ensemble de sommets donné, et la partie (b) montre l'arbre couvrant minimum T créé à partir de la racine a par ACM-PRIM. La partie (c) montre la manière dont les sommets sont visités par un parcours préfixe de T , et la partie (d) affiche la tournée correspondante, qui est celle retournée par TOURNÉE-VC-APPROCHÉE. La partie (e) affiche une tournée optimale, plus courte d'environ 23 %.

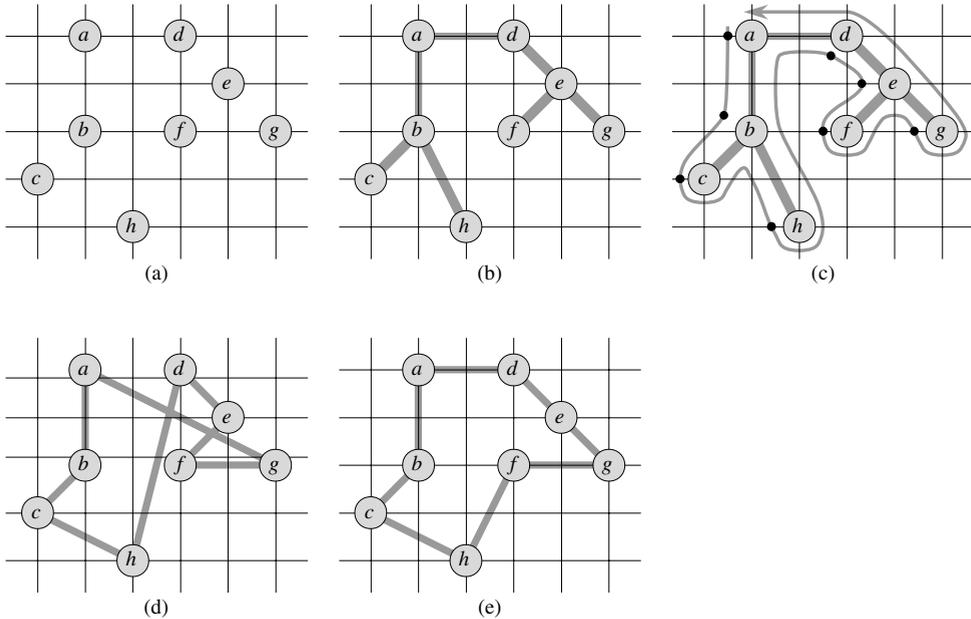


Figure 35.2 L'action de TOURNÉE-VC-APPROCHÉE. (a) L'ensemble de points donné, les points étant placés sur des sommets d'une grille à valeurs entières. Par exemple, f est situé à une unité à droite et deux unités plus haut que h . On utilise la distance euclidienne ordinaire comme fonction de coût entre deux points. (b) Un arbre couvrant minimum T sur ces points, calculé par ACM-PRIM. Le sommet a est la racine. Les sommets sont étiquetés de telle sorte qu'ils sont ajoutés à l'arbre principal par ACM-PRIM dans l'ordre alphabétique. (c) Un parcours de T , partant de a . Un parcours complet de l'arbre visite les sommets dans l'ordre $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. Un parcours préfixe de T affiche un sommet uniquement lors de la première rencontre, ce qui donne a, b, c, h, d, e, f, g . (d) Une tournée des sommets obtenue en visitant les sommets dans l'ordre donné par le parcours préfixe. C'est la tournée H retournée par TOURNÉE-VC-APPROCHÉE. Son coût total est d'environ 19,074. (e) Une tournée optimale H^* pour l'ensemble donné de sommets. Son coût total est d'environ 14,715.

D'après l'exercice 23.2.2, même avec une implémentation simple de ACM-PRIM, le temps d'exécution de TOURNÉE-VC-APPROCHÉE est $\Theta(S^2)$. Nous allons maintenant montrer que, si la fonction de coût d'une instance du problème du voyageur de commerce, vérifie l'inégalité triangulaire, alors TOURNÉE-VC-APPROCHÉE retourne une tournée dont le coût ne fait pas plus de deux fois le coût d'une tournée optimale.

Théorème 35.2 *TOURNÉE-VC-APPROCHÉE est un algorithme d'approximation avec une garantie de performance 2 à temps polynomial pour le problème du voyageur de commerce avec inégalité triangulaire.*

Démonstration : On a déjà montré que TOURNÉE-VC-APPROCHÉE s'exécute en temps polynomial. Soit H^* une tournée optimale pour l'ensemble de sommets donné. Puisqu'on obtient un arbre couvrant en supprimant des arêtes dans une tournée, le poids de l'arbre couvrant minimum T est un minorant du coût d'une tournée optimale ; donc, on a

$$c(T) \leq c(H^*) . \tag{35.4}$$

Un *parcours complet* de T liste les sommets quand ils sont visités pour la première fois, et aussi quand ils sont à nouveau traversés après une visite d'un sous-arbre. Soit W ce parcours. Le parcours complet, dans notre exemple, a pour résultat la liste

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

Puisque le parcours complet traverse chaque arête de T deux fois exactement, on a (en étendant de façon naturelle notre définition du coût c pour qu'elle s'applique aux multiensembles d'arêtes)

$$c(W) = 2c(T) . \quad (35.5)$$

Les équations (35.4) et (35.5) impliquent que

$$c(W) \leq 2c(H^*) , \quad (35.6)$$

et donc, le coût de W est inférieur à deux fois le coût d'une tournée optimale.

Malheureusement, W n'est en général pas une tournée, puisqu'il visite certains sommets plus d'une fois. Cependant, grâce à l'inégalité triangulaire, on peut supprimer dans W une visite à un sommet quelconque sans augmenter le coût. (Si un sommet v est supprimé de W entre les visites à u et w , l'ordre résultant spécifie d'aller directement de u à w .) En répétant cette opération, on peut supprimer de W toutes les visites de chaque sommet, sauf la première. Dans notre exemple, on se retrouve avec la liste réduite

$$a, b, c, h, d, e, f, g .$$

Cette liste est la même que celle obtenue via un parcours préfixe de l'arbre T . Soit H le cycle correspondant à ce parcours préfixe. C'est un cycle hamiltonien, puisque tout sommet est visité exactement une fois, et en fait c'est le cycle calculé par TOURNÉE-VC-APPROCHÉE. Puisque H est obtenue en supprimant des sommets du parcours complet W , on a

$$c(H) \leq c(W) . \quad (35.7)$$

En combinant les inégalités (35.6) et (35.7), on obtient $c(H) \leq 2c(H^*)$, ce qui achève la démonstration. \square

Malgré le sympathique ratio d'approximation donné par le théorème 35.2, TOURNÉE-VC-APPROCHÉE n'est généralement pas le meilleur choix en pratique pour ce problème. Il existe d'autres algorithmes d'approximation qui sont souvent meilleurs en pratique. (Voir les références de fin de chapitre.)

35.2.2 Problème général du voyageur de commerce

Si l'on élimine l'hypothèse selon laquelle la fonction coût c vérifie l'inégalité triangulaire, on ne peut pas trouver de bonnes tournées approchées à temps polynomial sauf si $P = NP$.

Théorème 35.3 *Si $P \neq NP$, alors pour toute constante $\rho \geq 1$, il n'existe aucun algorithme d'approximation à temps polynomial et à garantie de performance ρ pour le problème général du voyageur de commerce.*

Démonstration : La démonstration se fait par l'absurde. On suppose que, pour un certain nombre $\rho \geq 1$, il existe un algorithme d'approximation à temps polynomial B ayant le ratio ρ . Sans perte de généralité, on suppose que ρ est un entier, en l'arrondissant si nécessaire. Nous allons maintenant montrer comment utiliser B pour résoudre des instances du problème du cycle hamiltonien (défini à la section 34.2) en temps polynomial. Puisque le problème du cycle hamiltonien est NP-complet, d'après le théorème 34.13, le résoudre en temps polynomial implique $P = NP$, d'après le théorème 34.4

Soit $G = (S, A)$ une instance du problème du cycle hamiltonien. On souhaite déterminer efficacement si G contient un cycle hamiltonien en utilisant l'algorithme d'approximation B hypothétique. On transforme G en une instance du problème du voyageur de commerce de la façon suivante : Soit $G' = (S, A')$ le graphe complet sur S ; autrement dit,

$$A' = \{(u, v) : u, v \in S \text{ et } u \neq v\} .$$

On affecte un coût entier à chaque arête de A' comme suit :

$$c(u, v) = \begin{cases} 1 & \text{si } (u, v) \in A , \\ \rho |S| + 1 & \text{sinon .} \end{cases}$$

Les instances de G' et c peuvent être construites à partir d'une instance de G en temps polynomial en $|S|$ et $|A|$.

Considérons à présent le problème du voyageur de commerce (G', c) . Si le graphe G initial possède un cycle hamiltonien H , alors la fonction de coût c affecte à chaque arête de H un coût égal à 1, et (G', c) contient donc une tournée de coût $|S|$. D'un autre côté, si G ne contient pas de cycle hamiltonien, alors une tournée quelconque de G' utilise forcément une arête qui n'est pas dans A . Mais toute tournée utilisant une arête qui n'est pas dans A a un coût d'au moins

$$\begin{aligned} (\rho |V| + 1) + (|V| - 1) &= \rho |V| + |V| \\ &> \rho |V| . \end{aligned}$$

Puisque les arêtes qui ne sont pas dans G sont si chères, il existe un écart d'au moins $|S|$ entre le coût d'une tournée qui est un cycle hamiltonien de G (coût $|S|$) et le coût d'une quelconque autre tournée (coût d'au moins $\rho |S| + |S|$).

Que se passe-t-il si l'on applique l'algorithme d'approximation B au problème du voyageur de commerce (G', c) ? Comme on a la garantie que B retourne une tournée dont le coût n'est pas supérieur à ρ fois le coût d'une tournée optimale, si G contient un cycle hamiltonien, alors B retournera certainement ce cycle. Si G ne possède aucun cycle hamiltonien, alors B retournera une tournée dont le coût est supérieur à $\rho |S|$. On peut donc utiliser B pour résoudre le problème du cycle hamiltonien en temps polynomial. \square

La démonstration du théorème 35.3 est un exemple d'une technique générale de démonstration qu'un problème ne peut pas être bien approximé. Supposez que, étant donné un problème NP-difficile X , on puisse produire un problème de minimisation à temps polynomial Y tel que les instances « oui » de X correspondent aux instances de Y ayant une valeur au plus égale à k (pour un certain k), mais que les instances « non » de X correspondent aux instances de Y ayant une valeur supérieure à ρk .

Alors, on a montré que, à moins que $P = NP$, il n'y a pas d'algorithme d'approximation ρ à temps polynomial pour le problème Y .

Exercices

35.2.1 On suppose qu'un graphe non orienté complet $G = (S, A)$ ayant au moins 3 sommets a une fonction de coût c qui vérifie l'inégalité triangulaire. Prouver que $c(u, v) \geq 0$ pour tout $u, v \in S$.

35.2.2 Montrer comment on peut transformer en temps polynomial une instance du problème du voyageur de commerce en une autre instance dont la fonction coût vérifie l'inégalité triangulaire. Les deux instances doivent avoir le même ensemble de tournées optimales. Dire pourquoi ce type de transformation en temps polynomial ne contredit pas le théorème 35.3, en supposant que $P \neq NP$.

35.2.3 On considère l'*heuristique du point le plus proche* donnée ci-après, pour construire une tournée approchée pour le problème du voyageur de commerce. On commence avec un cycle trivial constitué d'un sommet unique choisi arbitrairement. À chaque étape, on identifie le sommet u qui n'appartient pas au cycle mais dont la distance à un sommet quelconque du cycle est minimale. Appelons v le sommet du cycle qui est le plus proche de u . On étend le cycle pour y inclure u , en insérant u juste après v . Cette action est répétée jusqu'à ce que tous les sommets se trouvent sur le cycle. Démontrer que cette heuristique retourne une tournée dont le coût total ne vaut pas plus de deux fois le coût d'une tournée optimale.

35.2.4 Le *problème du voyageur de commerce avec goulet d'étranglement* consiste à trouver un cycle hamiltonien tel que le coût de l'arête la plus coûteuse du cycle soit minimisé. En supposant que la fonction de coût vérifie l'inégalité triangulaire, montrer qu'il existe un algorithme d'approximation à temps polynomial et à ratio 3 pour ce problème. (*Conseil* : Montrer récursivement qu'on peut visiter tous les nœuds d'un arbre couvrant à goulet d'étranglement (voir problème 23-3) exactement une fois, en effectuant un parcours complet de l'arbre et en sautant des nœuds, mais sans sauter plus de 2 nœuds intermédiaires consécutifs. Montrer que l'arête la plus coûteuse d'un arbre couvrant à goulet d'étranglement a un coût qui est au plus égal au coût de l'arête la plus coûteuse d'un cycle hamiltonien à goulet d'étranglement.)

35.2.5 Supposez que les sommets d'une instance du problème du voyageur de commerce soient des points du plan et que la fonction de coût $c(u, v)$ soit la distance euclidienne entre les points u et v . Montrer qu'une tournée optimale ne se recoupe jamais elle-même.

35.3 PROBLÈME DE LA COUVERTURE D'ENSEMBLE

Le problème de la couverture d'ensemble est un problème d'optimisation qui modélise de nombreux problèmes de choix de ressources. Son problème de décision correspondant généralise le problème NP-complet de la couverture de sommets et est donc, lui aussi, NP-difficile. Toutefois, l'algorithme d'approximation développé pour

gérer le problème de la couverture de sommets ne s'applique pas ici, et il faut donc essayer d'autres approches. Nous examinerons une heuristique gloutonne simple, dotée d'une garantie de performance logarithmique. Autrement dit, quand la taille de l'instance augmente, la taille de la solution approchée peut croître, relativement à la taille d'une solution optimale. Comme la fonction logarithme croît plutôt lentement, cet algorithme d'approximation pourra néanmoins donner des résultats utiles.

Une instance (X, \mathcal{F}) du **problème de la couverture d'ensemble** consiste en un ensemble fini X et une famille \mathcal{F} de sous-ensembles de X , tels que tout élément de X appartient à au moins un sous-ensemble de \mathcal{F} :

$$X = \bigcup_{S \in \mathcal{F}} S.$$

On dit qu'un sous-ensemble $S \in \mathcal{F}$ **couvre** ses éléments. Le problème consiste à trouver un sous-ensemble de taille minimale $\mathcal{C} \subseteq \mathcal{F}$ dont les membres couvrent X tout entier :

$$X = \bigcup_{S \in \mathcal{C}} S. \quad (35.8)$$

On dit qu'un \mathcal{C} vérifiant l'équation (35.8) **couvre** X . La figure 35.3 illustre le problème de la couverture d'ensemble. La taille de \mathcal{C} est définie par le nombre d'ensembles qu'elle contient, pas par le nombre d'éléments individuels contenus dans ces ensembles. Sur la figure 35.3, la couverture d'ensemble minimale a la taille 3.

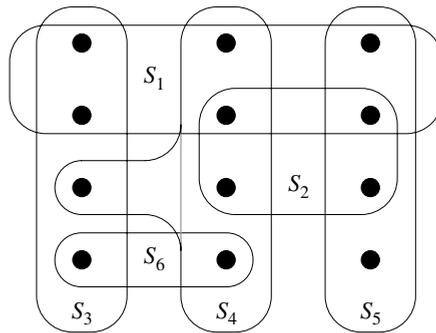


Figure 35.3 Une instance (X, \mathcal{F}) du problème de la couverture d'ensemble, où X est composé de 12 points noirs et $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Une couverture d'ensemble de taille minimale est $\mathcal{C} = \{S_3, S_4, S_5\}$. L'algorithme glouton produit une couverture de taille 4 en choisissant les ensembles S_1, S_4, S_5 et S_3 dans l'ordre.

Le problème de la couverture d'ensemble est une abstraction de nombreux problèmes combinatoires qui reviennent souvent. A titre d'exemple, supposons que X représente un ensemble de compétences nécessaires pour résoudre un problème, et qu'on dispose d'un ensemble de personnes disponibles pour travailler sur ce

problème. On souhaite créer un comité, composé du moins de personnes possible, tel que pour chaque compétence requise de X , un membre du comité possède cette compétence. Dans la version décisionnelle du problème de la couverture d'ensemble, on cherche à savoir s'il existe ou non une couverture de taille au plus égale à k , où k est un paramètre supplémentaire spécifié dans l'instance du problème. La version décisionnelle du problème est NP-complet, ce que demandera de montrer l'exercice 35.3.2.

a) Un algorithme d'approximation glouton

La méthode gloutonne fonctionne en choisissant, à chaque étape, l'ensemble S qui couvre le plus grand nombre d'éléments qui ne sont pas encore couverts.

```

COUVERTURE-ENSEMBLE-GLOUTON( $X, \mathcal{F}$ )
1   $U \leftarrow X$ 
2   $\mathcal{C} \leftarrow \emptyset$ 
3  tant que  $U \neq \emptyset$ 
4      faire choisir un  $S \in \mathcal{F}$  qui maximise  $|S \cap U|$ 
5           $U \leftarrow U - S$ 
6           $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$ 
7  retourner  $\mathcal{C}$ 

```

Dans l'exemple de la figure 35.3, COUVERTURE-ENSEMBLE-GLOUTON ajoute à \mathcal{C} les ensembles S_1, S_4, S_5 et S_3 dans l'ordre.

L'algorithme fonctionne de la manière suivante : l'ensemble U contient, à chaque étape, l'ensemble des éléments non encore couverts. L'ensemble \mathcal{C} contient la couverture en cours de construction. La ligne 4 constitue l'étape décisionnelle gloutonne. Un sous-ensemble S est choisi pour couvrir autant que faire se peut d'éléments non couverts (en cas d'égalité entre deux éléments, on en choisit un arbitrairement). Une fois que S est sélectionné, ses éléments sont supprimés de U et S est placé dans \mathcal{C} . A la fin de l'algorithme, l'ensemble \mathcal{C} contient une sous-famille de \mathcal{F} qui couvre X .

L'algorithme COUVERTURE-ENSEMBLE-GLOUTON peut être facilement implémenté pour s'exécuter en temps polynomial en $|X|$ et $|\mathcal{F}|$. Puisque le nombre d'itérations de la boucle des lignes 3–6 est majoré par $\min(|X|, |\mathcal{F}|)$ et que le corps de la boucle peut être implémenté pour s'exécuter en temps $O(|X| |\mathcal{F}|)$, il existe une implémentation pouvant s'exécuter en temps $O(|X| |\mathcal{F}| \min(|X|, |\mathcal{F}|))$. L'exercice 35.3.3 demandera de fournir un algorithme à temps linéaire.

b) Analyse

Nous allons à présent montrer que l'algorithme glouton retourne une couverture d'ensemble qui n'est pas beaucoup plus grande qu'une couverture d'ensemble optimale. Par commodité, le d ème nombre harmonique $H_d = \sum_{i=1}^d 1/i$ (voir section A.1) sera noté $H(d)$ dans ce chapitre. Comme condition aux limites, nous définirons $H(0) = 0$.

Théorème 35.4 COUVERTURE-ENSEMBLE-GLOUTON est un algorithme d'approximation fournissant une garantie de performance $\rho(n)$ à temps polynomial, avec

$$\rho(n) = H(\max \{|S| : S \in \mathcal{F}\}) .$$

Démonstration : Nous avons déjà montré que COUVERTURE-ENSEMBLE-GLOUTON s'exécute en temps polynomial. Pour montrer que c'est un algorithme d'approximation $\rho(n)$, on attribue un coût de 1 à chaque ensemble choisi par l'algorithme, on distribue ce coût sur les éléments couverts pour la première fois, puis on utilise ces coûts pour déduire la relation souhaitée entre la taille d'une couverture d'ensemble optimale \mathcal{C}^* et la taille de la couverture d'ensemble \mathcal{C} retournée par l'algorithme. Soit S_i le i ème sous-ensemble choisi par COUVERTURE-ENSEMBLE-GLOUTON ; l'algorithme génère un coût de 1 quand il ajoute S_i à \mathcal{C} . On répartit ce coût, dû au choix de S_i , équitablement entre les éléments couverts pour la première fois par S_i . Soit c_x le coût attribué à l'élément x , pour tout $x \in X$. Chaque élément se voit affecter un coût une seule fois, quand il est couvert pour la première fois. Si x est couvert pour la première fois par S_i , alors

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|} .$$

À chaque étape de l'algorithme, il y a assignation d'une unité de coût, de sorte que

$$|\mathcal{C}| = \sum_{x \in X} c_x .$$

Le coût assigné à la couverture optimale est

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x ,$$

et comme chaque $x \in X$ appartient à au moins un ensemble $S \in \mathcal{C}^*$, on a

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x .$$

En combinant les deux inégalités qui précèdent, on obtient

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x . \quad (35.9)$$

Le reste de la démonstration repose sur l'inégalité fondamentale suivante, que nous allons démontrer d'ici peu. Pour tout ensemble S appartenant à la famille \mathcal{F} ,

$$\sum_{x \in S} c_x \leq H(|S|) . \quad (35.10)$$

Des inégalités (35.9) et (35.10), il découle que

$$\begin{aligned} |\mathcal{C}| &\leq \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\leq |\mathcal{C}^*| \cdot H(\max \{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

ce qui démontre le théorème.

Tout ce qui reste à prouver, c'est l'inégalité (35.10). Soit un ensemble $S \in \mathcal{F}$ et $i = 1, 2, \dots, |\mathcal{C}|$, et soit

$$u_i = |S - (S_1 \cup S_2 \cup \dots \cup S_i)|$$

le nombre d'éléments de S qui ne sont pas encore couverts après que S_1, S_2, \dots, S_i ont été sélectionnés par l'algorithme. Soit $u_0 = |S|$ le nombre d'éléments de S , qui sont tous initialement non couverts. Soit k l'indice minimal tel que $u_k = 0$, de sorte que chaque élément de S est couvert par l'un au moins des ensembles S_1, S_2, \dots, S_k . Alors, $u_{i-1} \geq u_i$ et $u_{i-1} - u_i$ éléments de S sont couverts pour la première fois par S_i , avec $i = 1, 2, \dots, k$. Donc,

$$\sum_{x \in S} c_x = \sum_{i=1}^k (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}.$$

Observez que

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| &\geq |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \\ &= u_{i-1}, \end{aligned}$$

car le choix glouton de S_i garantit que S ne peut pas couvrir plus de nouveaux éléments que S_i (sinon, S aurait été sélectionné à la place de S_i). Par conséquent, on obtient

$$\sum_{x \in S} c_x \leq \sum_{i=1}^k (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Majorons cette quantité comme suit :

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^k (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\ &= \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\ &\leq \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && \text{(car } j \leq u_{i-1} \text{)} \\ &= \sum_{i=1}^k \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^k (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) && \text{(car les termes se télescopent)} \\ &= H(u_0) - H(0) \\ &= H(u_0) && \text{(car } H(0) = 0 \text{)} \\ &= H(|S|), \end{aligned}$$

ce qui achève la démonstration de l'inégalité (35.10). \square

Corollaire 35.5 COUVERTURE-ENSEMBLE-GLOUTON est un algorithme d'approximation à temps polynomial, avec garantie de performance $(\ln |X| + 1)$.

Démonstration : Découle de l'inégalité (A.14) et du théorème 35.4. □

Dans certaines applications, $\max \{|S| : S \in \mathcal{F}\}$ est une petite constante, et donc la solution retournée par COUVERTURE-ENSEMBLE-GLOUTON a une taille au plus égale à un petit multiple de la taille optimale. Voici un exemple d'un tel cas de figure : calcul d'une couverture de sommets approchée pour un graphe dont les sommets ont un degré au plus égal à 3. En pareil cas, la solution trouvée par COUVERTURE-ENSEMBLE-GLOUTON ne fait pas plus de $H(3) = 11/6$ fois la taille d'une solution optimale, garantie de performance qui est un peu meilleure que celle de COUVERTURE-SOMMET-APPROCHÉE.

Exercices

35.3.1 On considère chacun des mots suivants comme un ensemble de lettres : {aride, date, drain, héron, louche, nom, short, slalom, snob, thèse}. Montrer quelle est la couverture d'ensemble produite par COUVERTURE-ENSEMBLE-GLOUTON (en cas d'égalité, le choix glouton se fait en faveur du mot qui apparaît le premier dans le dictionnaire).

35.3.2 Montrer que la version décisionnelle du problème de la couverture d'ensemble est NP-complète en effectuant une réduction à partir du problème de la couverture de sommets.

35.3.3 Montrer comment implémenter COUVERTURE-ENSEMBLE-GLOUTON de telle sorte qu'elle s'exécute en temps $O(\sum_{S \in \mathcal{F}} |S|)$.

35.3.4 Montrer que la forme plus faible que voici du théorème 35.4 est trivialement vraie :

$$|\mathcal{C}| \leq |\mathcal{C}^*| \max \{|S| : S \in \mathcal{F}\} .$$

35.3.5 COUVERTURE-ENSEMBLE-GLOUTON peut retourner un certain nombre de solutions différentes, selon la façon dont on gère les cas d'égalité en ligne 4. Donner une procédure MAUVAISE-INSTANCE-COUVERTURE-ENSEMBLE(n) qui retourne une instance à n éléments du problème de la couverture d'ensemble pour laquelle, selon la façon dont les égalités sont gérées en ligne 4, COUVERTURE-ENSEMBLE-GLOUTON peut retourner un nombre de solutions différentes qui est exponentiel en n .

35.4 RANDOMISATION ET PROGRAMMATION LINÉAIRE

Dans cette section, nous allons étudier deux techniques qui sont utiles pour concevoir des algorithmes d'approximation : la randomisation et la programmation linéaire. Nous donnerons un algorithme randomisé simple pour une version optimisation de

la satisfaisabilité 3-CNF, puis nous emploierons la programmation linéaire pour faciliter la conception d'un algorithme d'approximation pour une version pondérée du problème de la couverture de sommets. Cette section ne fera qu'effleurer ces deux puissantes techniques. Reportez-vous aux notes de fin de chapitre pour des références bibliographiques concernant ces domaines.

a) *Algorithme d'approximation randomisé pour satisfaisabilité MAX-3-CNF*

De même qu'il y a des algorithmes randomisés qui calculent des solutions exactes, il y a des algorithmes randomisés qui calculent des solutions approchées. On dit qu'un algorithme randomisé pour un problème a une **garantie de performance** de $\rho(n)$ si, pour toute entrée de taille n , l'espérance du coût C de la solution produite par l'algorithme randomisé est éloigné au plus d'un facteur multiplicatif $\rho(n)$ du coût C^* d'une solution optimale :

$$\max \left(\frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n). \quad (35.11)$$

Un algorithme randomisé atteignant un ratio de $\rho(n)$ est dit **algorithme d'approximation $\rho(n)$ randomisé**. Autrement dit : un algorithme randomisé d'approximation est similaire à un algorithme d'approximation déterministe, sauf que le ratio d'approximation concerne une valeur moyenne.

Une instance particulière de la satisfaisabilité 3-CNF, telle que définie à la section 34.4, peut être ou non satisfaisable. Pour qu'elle soit satisfaisable, il faut qu'il y ait une assignation des variables telles que l'évaluation de chaque clause donne 1. Si une instance n'est pas satisfaisable, on peut avoir besoin de calculer sa « proximité » par rapport à la satisfaisabilité : on veut trouver une assignation des variables qui satisfait le maximum de clauses. Le problème de maximisation résultant porte le nom de **satisfaisabilité MAX-3-CNF**. L'entrée de la satisfaisabilité MAX-3-CNF est la même que celle de la satisfaisabilité 3-CNF, et le but est de retourner une assignation des variables qui maximise le nombre de clauses dont l'évaluation donne 1. Nous allons montrer que l'affectation aléatoire, à chaque variable, de 1 avec la probabilité $1/2$ et de 0 avec la probabilité $1/2$ est un algorithme d'approximation $8/7$ randomisé. Selon la définition de la satisfaisabilité 3-CNF (section 34.4) on exige que chaque clause se compose exactement de trois littéraux distincts. On supposera, en outre, qu'aucune clause ne contient à la fois une variable et sa négation. (L'exercice 35.4.1 vous demandera de supprimer cette dernière hypothèse.)

Théorème 35.6 *Soit une instance de la satisfaisabilité MAX-3-CNF à n variables x_1, x_2, \dots, x_n et m clauses ; alors, l'algorithme randomisé qui affecte indépendamment à chaque variable la valeur 1 avec une probabilité $1/2$ et la valeur 0 avec une probabilité $1/2$ est un algorithme d'approximation $8/7$ randomisé.*

Démonstration : Supposons que l'on ait affecté indépendamment à chaque variable la valeur 1 avec la probabilité $1/2$, et la valeur 0 avec la probabilité $1/2$. Pour

$i = 1, 2, \dots, n$, on définit la variable indicatrice

$$Y_i = I \{ \text{la clause } i \text{ est satisfaite} \} ,$$

de sorte que $Y_i = 1$ si l'un au moins des littéraux de la i ème clause a été mis à 1. Comme un littéral n'apparaît jamais plus d'une fois dans la même clause et que l'on a supposé qu'une même clause ne contient pas à la fois une variable et sa négation, les configurations des trois littéraux dans chaque clause sont indépendantes. Pour qu'une clause ne soit pas satisfaite, il faut que ses trois littéraux valent tous 0 et donc que $\Pr \{ \text{la clause } i \text{ n'est pas satisfaite} \} = (1/2)^3 = 1/8$. Par conséquent, $\Pr \{ \text{la clause } i \text{ est satisfaite} \} = 1 - 1/8 = 7/8$. D'après le lemme 5.1, on a donc $E[Y_i] = 7/8$. Soit Y le nombre de clauses satisfaites sur l'ensemble, de sorte que $Y = Y_1 + Y_2 + \dots + Y_m$. On a alors

$$\begin{aligned} E[Y] &= E \left[\sum_{i=1}^m Y_i \right] \\ &= \sum_{i=1}^m E[Y_i] \quad (\text{d'après la linéarité de l'espérance}) \\ &= \sum_{i=1}^m 7/8 \\ &= 7m/8 . \end{aligned}$$

Visiblement, m est un majorant du nombre de clauses satisfaites et donc la garantie de performance est au plus égale à $m/(7m/8) = 8/7$. \square

b) Approximation d'une couverture de sommets pondérée, à l'aide de la programmation linéaire

Dans le **problème de la couverture de sommets de poids minimal**, on se donne un graphe non orienté $G = (V, E)$ où chaque sommet $v \in V$ a un poids positif $w(v)$. Pour toute couverture de sommets $V' \subseteq V$, on définit le poids de la couverture $w(V') = \sum_{v \in V'} w(v)$. L'objectif est de trouver une couverture de sommets qui ait un poids minimal.

On ne peut pas appliquer l'algorithme employé pour la couverture de sommets non pondérée, ni faire appel à une solution randomisée ; ces deux méthodes risquent de donner des solutions rien moins qu'optimales. Nous allons, toutefois, calculer un minorant du poids de la couverture de sommets de poids minimal, et ce en utilisant un programme linéaire. Nous « arrondirons » ensuite cette solution et nous nous en servirons pour obtenir une couverture de sommets.

Supposons que l'on associe une variable $x(v)$ à chaque sommet $v \in V$, en exigeant que $x(v) \in \{0, 1\}$ pour tout $v \in V$. Nous interprétons $x(v) = 1$ comme signifiant que v est dans la couverture de sommets, et $x(v) = 0$ comme signifiant le contraire. Nous pouvons alors écrire la contrainte que, pour toute arête (u, v) , l'un au moins de u et v est forcément dans la couverture de sommets sous la forme $x(u) + x(v) \geq 1$. Cette façon de voir les choses donne naissance au **programme entier 0-1** suivant,

concernant le calcul d'une couverture de sommets de poids minimal :

$$\text{minimiser} \quad \sum_{v \in V} w(v)x(v) \quad (35.12)$$

sous les contraintes

$$x(u) + x(v) \geq 1 \quad \text{pour tout } (u, v) \in E \quad (35.13)$$

$$x(v) \in \{0, 1\} \quad \text{pour tout } v \in V. \quad (35.14)$$

D'après l'exercice 34.5.2, nous savons que le simple fait de trouver des valeurs de $x(v)$ qui vérifient (35.13) et (35.14) est NP-difficile, de sorte que cette formulation n'est pas immédiatement utile. Supposons, toutefois, que l'on supprime la contrainte que $x(v) \in \{0, 1\}$ et qu'on la remplace par $0 \leq x(v) \leq 1$. On obtient alors le programme linéaire suivant, connu sous le nom de **relaxé programmation linéaire** :

$$\text{minimiser} \quad \sum_{v \in V} w(v)x(v) \quad (35.15)$$

sous les contraintes

$$x(u) + x(v) \geq 1 \quad \text{pour tout } (u, v) \in E \quad (35.16)$$

$$x(v) \leq 1 \quad \text{pour tout } v \in V \quad (35.17)$$

$$x(v) \geq 0 \quad \text{pour tout } v \in V. \quad (35.18)$$

Toute solution réalisable du programme entier 0-1 défini aux lignes (35.12)–(35.14) est aussi une solution réalisable du programme linéaire défini aux lignes (35.15)–(35.18). Par conséquent, une solution optimale du programme linéaire est un minorant de la solution optimale du programme entier 0-1, et partant un minorant d'une solution optimale du problème de la couverture de sommets de poids minimal.

La procédure suivante emploie la solution du programme linéaire précédent pour construire une solution approchée du problème de la couverture de sommets de poids minimal :

COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE(G, w)

- 1 $C \leftarrow \emptyset$
- 2 calculer \bar{x} , solution optimale du programme linéaire des lignes (35.15)–(35.18)
- 3 **pour** tout $v \in V$
- 4 **faire si** $\bar{x}(v) \geq 1/2$
- 5 **alors** $C \leftarrow C \cup \{v\}$
- 6 **retourner** C

Voici comment fonctionne COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE. La ligne 1 initialise la couverture de sommets de façon qu'elle soit vide. La ligne 2 formule le programme linéaire des lignes (35.15)–(35.18), puis résout ce programme. Une solution optimale donne à chaque sommet v une valeur associée $\bar{x}(v)$, avec $0 \leq \bar{x}(v) \leq 1$. On utilise cette valeur pour guider le choix des sommets à ajouter à la couverture C aux lignes 3–5. Si $\bar{x}(v) \geq 1/2$, on ajoute v à C ; sinon, on n'ajoute pas.

En fait, on « arrondit » chaque variable de la solution du programme linéaire à 0 ou à 1 afin d'obtenir une solution au programme entier 0-1, aux lignes (35.12)–(35.14). Enfin, la ligne 6 retourne la couverture C .

Théorème 35.7 *L'algorithme COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE est un algorithme d'approximation 2 à temps polynomial pour le problème de la couverture de sommets de poids minimal.*

Démonstration : Comme il y a un algorithme à temps polynomial pour résoudre le programme linéaire en ligne 2 et que la boucle **pour** des lignes 3–5 s'exécute en temps polynomial, COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE est un algorithme à temps polynomial.

Montrons maintenant que COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE est un algorithme d'approximation 2. Soit C^* une solution optimale du problème de la couverture de sommets de poids minimal et soit z^* la valeur d'une solution optimale du programme linéaire des lignes (35.15)–(35.18). Comme une couverture de sommets optimale est une solution réalisable du programme linéaire, z^* est forcément un minorant de $w(C^*)$, c'est à dire que

$$z^* \leq w(C^*) . \quad (35.19)$$

Nous affirmons ensuite que, en arrondissant les valeurs des variables $\bar{x}(v)$, nous produisons un ensemble C qui est une couverture de sommets et qui vérifie $w(C) \leq 2z^*$. Pour vérifier que C est une couverture, prenons une arête $(u, v) \in E$. D'après la contrainte (35.16), on sait que $x(u) + x(v) \geq 1$, ce qui implique que l'un au moins de $\bar{x}(u)$ et $\bar{x}(v)$ vaut au moins $1/2$. Par conséquent, l'un au moins de u et v sera inclus dans la couverture, et donc chaque arête sera couverte.

Considérons maintenant le poids de la couverture. On a

$$\begin{aligned} z^* &= \sum_{v \in V} w(v) \bar{x}(v) \\ &\geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \bar{x}(v) \\ &\geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\ &= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in C} w(v) \\ &= \frac{1}{2} w(C) . \end{aligned} \quad (35.20)$$

En combinant les inégalités (35.19) et (35.20), on obtient

$$w(C) \leq 2z^* \leq 2w(C^*) ,$$

ce qui entraîne que COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE est un algorithme d'approximation ayant une garantie de performance 2. \square

Exercices

35.4.1 Montrer que, même si l'on permet à une clause de contenir à la fois une variable et sa négation, affecter aléatoirement à chaque variable la valeur 1 avec la probabilité $1/2$ et la valeur 0 avec la probabilité $1/2$, cela donne encore un algorithme d'approximation $8/7$ randomisé.

35.4.2 Le *problème de la satisfaisabilité MAX-CNF* ressemble au problème de la satisfaisabilité MAX-3-CNF, sauf qu'il n'oblige pas chaque clause à avoir exactement 3 littéraux. Donner un algorithme d'approximation 2 randomisé pour le problème de la satisfaisabilité MAX-CNF.

35.4.3 Dans le problème MAX-CUT, on a un graphe non orienté non pondéré $G = (V, E)$. On définit une coupe $(S, V - S)$ comme au chapitre 23 et on définit le *poids* d'une coupe comme étant le nombre d'arêtes traversant la coupe. Le but est de trouver une coupe de poids maximal. On suppose que, pour chaque sommet v , on place aléatoirement et indépendamment v dans S avec la probabilité $1/2$, et dans $V - S$ avec la probabilité $1/2$. Montrer que cet algorithme est un algorithme d'approximation 2 randomisé.

35.4.4 Montrer que les contraintes de la ligne (35.17) sont redondantes, en ce sens que, si on les supprime du programme linéaire des lignes (35.15)–(35.18), toute solution optimale du programme linéaire résultant vérifie forcément $x(v) \leq 1$ pour tout $v \in V$.

35.5 PROBLÈME DE LA SOMME DE SOUS-ENSEMBLE

Une instance du problème de la somme de sous-ensemble est une paire (S, t) , où S est un ensemble $\{x_1, x_2, \dots, x_n\}$ d'entiers positifs et t est un entier positif. Ce problème de décision consiste à savoir s'il existe un sous-ensemble de S dont la somme des éléments a pour valeur t . Ce problème est NP-complet (voir section 34.5.5).

Le problème d'optimisation associé à ce problème de décision survient dans des applications pratiques. Dans le problème d'optimisation, on souhaite trouver un sous-ensemble de $\{x_1, x_2, \dots, x_n\}$ dont la somme est aussi grande que possible, mais sans dépasser t . Par exemple, on pourrait disposer d'un camion dont la charge maximale est t tonnes, et de n boîtes différentes à convoier, la i ème pesant x_i kilogrammes. On souhaite remplir le camion au maximum, sans excéder le poids limite.

Dans cette section, nous présenterons un algorithme à temps exponentiel pour ce problème d'optimisation, puis montrerons comment le modifier pour qu'il devienne un schéma d'approximation à temps entièrement polynomial. (Rappelez-vous qu'un schéma d'approximation à temps entièrement polynomial a un temps d'exécution qui polynomial en $1/\varepsilon$ et en n , où n désigne la taille de l'entrée.)

a) Un algorithme à temps exponentiel

Supposez que nous calculions, pour chaque sous-ensemble S' de S , la somme des éléments de S' , puis que nous sélectionnions, parmi les sous-ensembles dont la

somme ne dépasse pas t , celui dont la somme est la plus proche de t . Visiblement, cet algorithme retourne la solution optimale, mais il peut prendre un temps exponentiel. Pour implémenter cet algorithme, nous pourrions utiliser une procédure itérative qui, à l'itération i , calcule les sommes de tous les sous-ensembles de $\{x_1, x_2, \dots, x_i\}$, en prenant comme point de départ les sommes de tous les sous-ensembles de $\{x_1, x_2, \dots, x_{i-1}\}$. En procédant ainsi, nous nous apercevons que, dès qu'un certain sous-ensemble S' a une somme supérieure à t , il n'y a plus de raison de le conserver, vu qu'aucun sur-ensemble de S' ne peut être la solution optimale. Nous allons maintenant donner une implémentation de cette stratégie.

La procédure SOMME-EXACTE-SOUS-ENSEMBLE prend en entrée un ensemble $S = \{x_1, x_2, \dots, x_n\}$ et une valeur cible t ; nous verrons le pseudo code dans un moment. Cette procédure calcule itérativement L_i , liste des sommes de tous les sous-ensembles de $\{x_1, \dots, x_i\}$ qui ne dépassent pas t , puis retourne la valeur maximale de L_n .

Si L est une liste d'entiers positifs et si x est un entier positif, on appelle $L + x$ la liste des entiers déduits de L en augmentant chaque élément de L de la valeur x . Par exemple, si $L = \langle 1, 2, 3, 5, 9 \rangle$, alors $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. On utilise également cette notation pour les ensembles, de sorte que

$$S + x = \{s + x : s \in S\} .$$

On fait appel à une procédure auxiliaire FUSIONNER-LISTES(L, L') qui retourne la liste triée résultant de la fusion de deux listes triées L et L' données en entrée. Comme la procédure FUSIONNER utilisée pour le tri par fusion (section 2.3.1), FUSIONNER-LISTES s'exécute en temps $O(|L| + |L'|)$. (Nous ne donnerons pas le pseudo code de FUSIONNER-LISTES.)

SOMME-EXACTE-SOUS-ENSEMBLE(S, t)

```

1   $n \leftarrow |S|$ 
2   $L_0 \leftarrow \langle 0 \rangle$ 
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire  $L_i \leftarrow$  FUSIONNER-LISTES( $L_{i-1}, L_{i-1} + x_i$ )
5          supprimer de  $L_i$  tout élément supérieur à  $t$ 
6  retourner le plus grand élément de  $L_n$ 
```

Pour comprendre le fonctionnement de SOMME-EXACTE-SOUS-ENSEMBLE, notons P_i l'ensemble de toutes les valeurs pouvant être obtenues en choisissant un sous-ensemble (éventuellement vide) de $\{x_1, x_2, \dots, x_i\}$ et en faisant la somme de tous ses éléments. Par exemple, si $S = \{1, 4, 5\}$, alors

$$\begin{aligned} P_1 &= \{0, 1\} , \\ P_2 &= \{0, 1, 4, 5\} , \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} . \end{aligned}$$

Étant donnée l'identité

$$P_i = P_{i-1} \cup (P_{i-1} + x_i), \quad (35.21)$$

on peut démontrer par récurrence sur i (voir exercice 35.5.1) que la liste L_i est une liste triée contenant tous les éléments de P_i dont la valeur n'est pas supérieure à t . Comme la longueur de L_i peut valoir jusqu'à 2^i , SOMME-EXACTE-SOUS-ENSEMBLE est, en général, un algorithme à temps exponentiel, bien que son temps d'exécution soit polynomial dans les cas particuliers où t est polynomial en $|S|$ ou tous les nombres de S sont bornés par un polynôme en $|S|$.

b) Un schéma d'approximation à temps entièrement polynomial

On peut élaborer un schéma d'approximation à temps entièrement polynomial pour le problème de la somme de sous-ensemble en « épurer » chaque liste L_i après sa création. L'idée ici est que, si deux valeurs de L sont proches l'une de l'autre, point n'est besoin de conserver les deux quand on veut simplement trouver une solution approchée. Plus précisément, on emploie un paramètre d'épuration δ tel que $0 < \delta < 1$. **Épurer** une liste L de δ signifie ceci : supprimer le maximum d'éléments dans L de telle façon que, si L' est le résultat de l'épuration de L , alors, pour tout élément y supprimé dans L , il existe encore un élément $z \leq y$ dans L' tel que

$$\frac{y}{1 + \delta} \leq z \leq y. \quad (35.22)$$

On peut voir ce z comme le « représentant » de y dans la nouvelle liste L' . Chaque y est représenté par un z satisfaisant à l'inégalité 35.22. Par exemple, si $\delta = 0,1$ et

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle,$$

alors on peut épurer L pour obtenir

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle,$$

où la valeur supprimée 11 est représentée par 10, les valeurs supprimées 21 et 22 sont représentées par 20, et la valeur supprimée 24 est représentée par 23. Comme chaque élément de la version épurée de la liste appartient aussi à la version initiale de la liste, l'épuration peut diminuer de façon importante le nombre d'éléments gérés, tout en conservant dans la liste une valeur représentative approchée (et légèrement plus petite) pour chaque élément supprimé.

La procédure ci-après épure une liste $L = \langle y_1, y_2, \dots, y_m \rangle$ en temps $\Theta(m)$, L et δ étant donnés, en supposant que L est triée par ordre monotone croissant. Le résultat de la procédure est une liste épurée et triée.

```

ÉPURER( $L, \delta$ )
1   $m \leftarrow |L|$ 
2   $L' \leftarrow \langle y_1 \rangle$ 
3   $dernier \leftarrow y_1$ 
4  pour  $i \leftarrow 2$  à  $m$ 
5      faire si  $y_i > dernier \cdot (1 + \delta)$   $\triangleright y_i \geq dernier$  car  $L$  est triée
6          alors ajouter  $y_i$  à la fin de  $L'$ 
7           $dernier \leftarrow y_i$ 
8  retourner  $L'$ 

```

Les éléments de L sont analysés dans l'ordre croissant, et un nombre n 'est placé dans la liste L' que s'il est le premier élément de L ou s'il ne peut pas être représenté par le nombre le plus récemment placé dans L' .

Étant donnée la procédure ÉPURER, on peut construire notre schéma d'approximation de la manière suivante. Cette procédure prend en entrée un ensemble $S = \{x_1, x_2, \dots, x_n\}$ de n entiers (dans un ordre quelconque), un entier cible t et un « paramètre d'approximation » ε , avec

$$0 < \varepsilon < 1. \quad (35.23)$$

Elle retourne une valeur z d'un facteur multiplicatif d'au plus $(1 + \varepsilon)$ de la solution optimale.

```

SOMME-SOUS-ENSEMBLE-APPROCHÉE( $S, t, \varepsilon$ )
1   $n \leftarrow |S|$ 
2   $L_0 \leftarrow \langle 0 \rangle$ 
3  pour  $i \leftarrow 1$  à  $n$ 
4      faire  $L_i \leftarrow \text{FUSIONNER-LISTES}(L_{i-1}, L_{i-1} + x_i)$ 
5           $L_i \leftarrow \text{ÉPURER}(L_i, \varepsilon/2n)$ 
6          supprimer de  $L_i$  chaque élément qui est supérieur à  $t$ 
7  soit  $z^*$  la valeur maximale de  $L_n$ 
8  retourner  $z^*$ 

```

La ligne 2 initialise la liste L_0 pour qu'elle ne contienne que l'élément 0. La boucle **pour** des lignes 3–6 a pour effet de calculer L_i en tant que liste triée contenant une version convenablement épurée de l'ensemble P_i , avec suppression de tous les éléments supérieurs à t . Comme L_i est créée à partir de L_{i-1} , il faut garantir que l'épuration répétée n'introduit pas une trop grande perte de précision. Nous verrons bientôt que SOMME-SOUS-ENSEMBLE-APPROCHÉE retourne une approximation correcte si cette approximation existe.

À titre d'exemple, supposons qu'on ait l'instance

$$L = \langle 104, 102, 201, 101 \rangle$$

avec $t = 308$ et $\varepsilon = 0,40$. Le paramètre d'épuration δ vaut $\varepsilon/8 = 0,05$. SOMME-SOUS-ENSEMBLE-APPROCHÉE calcule les valeurs suivantes sur les lignes indiquées :

$$\text{ligne 2 : } L_0 = \langle 0 \rangle ,$$

$$\text{ligne 4 : } L_1 = \langle 0, 104 \rangle ,$$

$$\text{ligne 5 : } L_1 = \langle 0, 104 \rangle ,$$

$$\text{ligne 6 : } L_1 = \langle 0, 104 \rangle ,$$

$$\text{ligne 4 : } L_2 = \langle 0, 102, 104, 206 \rangle ,$$

$$\text{ligne 5 : } L_2 = \langle 0, 102, 206 \rangle ,$$

$$\text{ligne 6 : } L_2 = \langle 0, 102, 206 \rangle ,$$

$$\text{ligne 4 : } L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle ,$$

$$\text{ligne 5 : } L_3 = \langle 0, 102, 201, 303, 407 \rangle ,$$

$$\text{ligne 6 : } L_3 = \langle 0, 102, 201, 303 \rangle ,$$

$$\text{ligne 4 : } L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle ,$$

$$\text{ligne 5 : } L_4 = \langle 0, 101, 201, 302, 404 \rangle ,$$

$$\text{ligne 6 : } L_4 = \langle 0, 101, 201, 302 \rangle .$$

L'algorithme retourne la réponse $z^* = 302$ qui s'écarte effectivement de moins de $\varepsilon = 40\%$ par rapport à la réponse optimale $307 = 104 + 102 + 101$; en fait, elle s'en écarte de moins de 2% .

Théorème 35.8 SOMME-SOUS-ENSEMBLE-APPROCHÉE est un schéma d'approximation à temps entièrement polynomial pour le problème de la somme de sous-ensemble.

Démonstration : L'épuration de L_i en ligne 5 et la suppression dans L_i de tout élément qui est plus grand que t , tout cela maintient la propriété que tout élément de L_i est aussi membre de P_i . Donc, la valeur z^* retournée en ligne 8 est bien la somme d'un certain sous-ensemble de S . Soit $y^* \in P_n$ une solution optimale du problème de la somme de sous-ensemble. Alors, d'après la ligne 6, on sait que $z^* \leq y^*$. D'après l'inégalité (35.1), on a besoin de montrer que $y^*/z^* \leq 1 + \varepsilon$. On doit aussi montrer que le temps d'exécution de cet algorithme est polynomial en $1/\varepsilon$ et en la taille de l'entrée.

Une récurrence sur i permet de montrer que, pour tout élément y de P_i qui est au plus égal à t , il existe $z \in L_i$ tel que

$$\frac{y}{(1 + \varepsilon/2n)^i} \leq z \leq y \quad (35.24)$$

(voir exercice 35.5.2). L'inégalité (35.24) est forcément vraie pour $y^* \in P_n$, et donc il existe $z \in L_n$ tel que

$$\frac{y^*}{(1 + \varepsilon/2n)^n} \leq z \leq y^* ,$$

D'où

$$\frac{y^*}{z} \leq \left(1 + \frac{\varepsilon}{2n}\right)^n. \quad (35.25)$$

Comme il existe $z \in L_n$ qui vérifie l'inégalité (35.25), l'inégalité est vraie pour z^* , qui est l'élément maximal de L_n ; donc

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\varepsilon}{2n}\right)^n. \quad (35.26)$$

Reste à montrer que $y^*/z^* \leq 1 + \varepsilon$. Pour ce faire, nous allons montrer que $(1 + \varepsilon/2n)^n \leq 1 + \varepsilon$. D'après l'équation (3.13), on a $\lim_{n \rightarrow \infty} (1 + \varepsilon/2n)^n = e^{\varepsilon/2}$. Comme on peut aussi prouver que

$$\frac{d}{dn} \left(1 + \frac{\varepsilon}{2n}\right)^n > 0, \quad (35.27)$$

la fonction $(1 + \varepsilon/2n)^n$ croît avec n quand il tend vers sa limite de $e^{\varepsilon/2}$, et l'on a

$$\begin{aligned} \left(1 + \frac{\varepsilon}{2n}\right)^n &\leq e^{\varepsilon/2} \\ &\leq 1 + \varepsilon/2 + (\varepsilon/2)^2 \quad (\text{d'après l'inégalité (3.12)}) \\ &\leq 1 + \varepsilon \quad (\text{d'après l'inégalité (35.23)}) \end{aligned} \quad (35.28)$$

En combinant les inégalités (35.26) et (35.28), on parachève l'analyse de la garantie de performance.

Pour montrer que SOMME-SOUS-ENSEMBLE-APPROCHÉE est un schéma d'approximation à temps entièrement polynomial, on calcule une borne pour la longueur de L_i . Après épuration, des éléments consécutifs z et z' de L_i doivent vérifier la relation $z'/z > 1 + \varepsilon/2n$. En d'autres termes, ils doivent différer d'un facteur qui est au moins égal à $1 + \varepsilon/2n$. Chaque liste contient donc la valeur 0, éventuellement la valeur 1, et jusqu'à $\lceil \log_{1+\varepsilon/2n} t \rceil$ autres valeurs. Le nombre d'éléments de chaque liste L_i vaut au plus

$$\begin{aligned} \log_{1+\varepsilon/2n} t + 2 &= \frac{\ln t}{\ln(1 + \varepsilon/2n)} + 2 \\ &\leq \frac{2n(1 + \varepsilon/2n) \ln t}{\varepsilon} + 2 \quad (\text{d'après l'inégalité (3.16)}) \\ &\leq \frac{4n \ln t}{\varepsilon} + 2 \quad (\text{d'après l'inégalité (35.23)}) \end{aligned}$$

Cette borne est polynomiale par rapport à la taille de l'entrée (qui est le nombre de bits $\lg t$ nécessaire pour représenter t plus le nombre de bits nécessaire pour représenter l'ensemble S , valeur elle-même polynomiale en n) et par rapport à $1/\varepsilon$. Comme le temps d'exécution de SOMME-SOUS-ENSEMBLE-APPROCHÉE est polynomial par rapport aux longueurs des L_i , SOMME-SOUS-ENSEMBLE-APPROCHÉE est un schéma d'approximation à temps entièrement polynomial. \square

Exercices

35.5.1 Prouver l'équation (35.21). Montrer ensuite que, après exécution de la ligne 5 de SOMME-EXACTE-SOUS-ENSEMBLE, L_i est une liste triée qui contient tous les éléments de P_i dont la valeur n'est pas supérieure à t .

35.5.2 Prouver l'inégalité (35.24).

35.5.3 Prouver l'inégalité (35.27).

35.5.4 Comment pourrait-on modifier le schéma d'approximation présenté dans cette section pour trouver une bonne approximation de la plus petite valeur supérieure à t qui est la somme d'un certain sous-ensemble de la liste donnée en entrée ?

PROBLÈMES

35.1. Remplir des boîtes (BIN packing)

Supposons qu'on se donne un ensemble de n objets, où la taille s_i du i ème objet vérifie $0 < s_i < 1$. On souhaite ranger tous les objets dans un minimum de boîtes de taille unitaire. Chaque boîte peut contenir n'importe quel sous-ensemble d'objets dont la taille globale n'excède pas 1.

- a.** Démontrer que le problème consistant à déterminer le nombre minimal de boîtes nécessaires est NP-difficile. (*conseil* : Réduire à partir du problème de la somme de sous-ensemble.)

L'algorithme *first-fit* (premier apte) prend chaque objet l'un après l'autre et le place dans la première boîte qui peut l'accueillir. Soit $S = \sum_{i=1}^n s_i$.

- b.** Montrer que le nombre optimal de boîtes nécessaires est au moins égal à $\lceil S \rceil$.
- c.** Montrer que l'algorithme first-fit (premier apte) laisse au plus une boîte remplie à moins de la moitié.
- d.** Démontrer que le nombre de boîtes utilisées par l'algorithme first-fit (premier apte) n'est jamais supérieur à $\lceil 2S \rceil$.
- e.** Établir une garantie de performance de 2 pour l'algorithme first-fit (premier apte).
- f.** Donner une implémentation efficace de l'algorithme first-fit (premier apte) et analyser son temps d'exécution.

35.2. Approximation de la taille d'une clique maximale

Soit $G = (S, A)$ un graphe non orienté. Pour un $k \geq 1$ quelconque, on définit $G^{(k)}$ comme étant le graphe non orienté $(S^{(k)}, A^{(k)})$, où $S^{(k)}$ est l'ensemble de tous les k -uplets ordonnés de sommets de S et où $A^{(k)}$ est défini ainsi : (v_1, v_2, \dots, v_k) est adjacent à (w_1, w_2, \dots, w_k) si et seulement si, pour tout i vérifiant $1 \leq i \leq k$, on a soit le sommet v_i adjacent à w_i dans G soit $v_i = w_i$.

- a.** Démontrer que la taille de la clique maximale de $G^{(k)}$ est égale à la puissance k ème de la taille de la clique maximale de G .

- b.* Montrer que, s'il existe un algorithme d'approximation à ratio constant qui puisse trouver une clique de taille maximale, alors il existe un schéma d'approximation à temps entièrement polynomial pour le problème.

35.3. Problème de la couverture d'ensemble pondérée

Supposons qu'on généralise le problème de la couverture d'ensemble de sorte que chaque ensemble S_i de la famille \mathcal{F} ait un coût associé w_i et que le poids d'une couverture \mathcal{C} soit $\sum_{S_i \in \mathcal{C}} w_i$. On souhaite déterminer une couverture de poids minimal. (La section 35.3 gère le cas où $w_i = 1$ pour tout i .)

Montrer que l'algorithme glouton de la couverture d'ensemble peut être généralisé de façon naturelle pour fournir une solution approchée pour toute instance du problème de la couverture d'ensemble pondérée. Montrer que votre algorithme a une garantie de performance égale à $H(d)$, où d est la taille maximale d'un ensemble S_i .

35.4. Couplage maximum

Rappelons-nous que, pour un graphe non orienté G , un couplage est un ensemble d'arêtes tel qu'il n'y ait jamais deux arêtes de l'ensemble qui soient incidents au même sommet. À la section 26.3, nous avons vu comment trouver un couplage maximum dans un graphe biparti. Dans ce problème, vous allons étudier les couplages dans les graphes non orientés en général (c'est à dire que les graphes ne seront plus forcément bipartis).

- a.* Un *couplage maximal* est un couplage qui n'est pas un sous-ensemble propre d'un autre couplage. Montrer qu'un couplage maximal n'est pas forcément un couplage maximum en exhibant un graphe non orienté G et un couplage maximal M de G qui n'est pas un couplage maximum. (Il existe un graphe de ce genre ayant seulement quatre sommets.)
- b.* Soit un graphe non orienté $G = (V, E)$. Donner un algorithme glouton à temps $O(E)$ pour trouver un couplage maximal de G .

Dans ce problème, nous allons nous pencher sur un algorithme d'approximation à temps polynomial pour couplage maximum. Alors que l'algorithme de couplage maximum le plus rapide qui soit connu s'exécute en temps supra-linéaire (mais polynomial), l'algorithme d'approximation donné ici s'exécutera en temps linéaire. Vous montrerez que l'algorithme glouton à temps linéaire pour couplage maximal donné à la partie (b) est un algorithme d'approximation avec une garantie de performance 2 pour couplage maximum.

- c.* Montrer que la taille d'un couplage maximum de G est un minorant de la taille d'une couverture de sommets de G .
- d.* Soit un couplage maximal M de $G = (V, E)$. Soit

$$T = \{v \in V : \text{une certaine arête de } M \text{ est incidente à } v\} .$$

Que peut-on dire du sous-graphe de G induit par les sommets de G qui ne sont pas dans T ?

- e. Conclure de la partie (d) que $2|M|$ est la taille d'une couverture de sommets de G .
- f. En utilisant les parties (c) et (e), prouver que l'algorithme glouton de la partie (b) est un algorithme d'approximation 2 pour couplage maximum.

35.5. Ordonnement pour machines parallèles

Dans le *problème de l'ordonnement pour machines parallèles*, on se donne n tâches J_1, J_2, \dots, J_n , la tâche J_k ayant une durée de traitement p_k . On a aussi m machines identiques M_1, M_2, \dots, M_m . Un *ordonnement* spécifie, pour chaque tâche J_k , la machine sur laquelle elle s'exécutera et la période pendant laquelle elle s'exécutera. Chaque tâche J_k doit s'exécuter sur une certaine machine M_i pendant p_k unités de temps consécutives, et pendant cette période aucune autre tâche ne peut s'exécuter sur M_i . Soit C_k la *date d'achèvement* de la tâche J_k , c'est-à-dire la date à laquelle la tâche J_k aura fini de s'exécuter. Étant donné un ordonnancement, on définit $C_{\max} = \max_{1 \leq j \leq n} C_j$ comme étant la *durée globale* (makespan) de l'ordonnement. On veut trouver un ordonnancement ayant une durée globale minimale.

Par exemple, supposons que l'on ait deux machines M_1 et M_2 et quatre tâches J_1, J_2, J_3, J_4 , avec $p_1 = 2, p_2 = 12, p_3 = 4$ et $p_4 = 5$. Alors, un ordonnancement possible fait s'exécuter, sur la machine M_1 , la tâche J_1 suivie de la tâche J_2 , et sur la machine M_2 , la tâche J_4 suivie de la tâche J_3 . Pour cet ordonnancement, $C_1 = 2, C_2 = 14, C_3 = 9, C_4 = 5$ et $C_{\max} = 14$. Un ordonnancement optimal fait s'exécuter J_2 sur la machine M_1 , et J_1, J_3 et J_4 sur la machine M_2 . Pour cet ordonnancement, $C_1 = 2, C_2 = 12, C_3 = 6, C_4 = 11$ et $C_{\max} = 12$.

Étant donné un problème d'ordonnement pour machines parallèles, soit C_{\max}^* la durée globale d'un ordonnancement optimal.

- a. Montrer que la durée globale optimale est au moins aussi grand que la durée maximale d'exécution d'une tâche, c'est à dire que

$$C_{\max}^* \geq \max_{1 \leq k \leq n} p_k .$$

- b. Montrer que le délai global optimal est au moins aussi grand que la charge machine moyenne, c'est à dire que

$$C_{\max}^* \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k .$$

Supposons que l'on emploie l'algorithme glouton suivant pour l'ordonnement sur machines parallèles : chaque fois qu'une machine est inactive, ordonner une tâche qui n'a pas encore été ordonnancée.

- c. Écrire un pseudo code qui implémente cet algorithme glouton. Quel est le temps d'exécution de cet algorithme ?

d. Pour l'ordonnancement retourné par l'algorithme glouton, montrer que

$$C_{\max} \leq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k .$$

En conclure que cet algorithme est un algorithme d'approximation avec une garantie de performance 2 à temps polynomial.

NOTES

Bien que l'on connaisse depuis des milliers d'années des méthodes qui ne calculent pas nécessairement des solutions exactes (par exemple, des méthodes pour approximer la valeur de π), la notion d'algorithme d'approximation est bien plus récente. Hochbaum attribue à Garey, Graham, et Ullman [109] et Johnson [166] la formalisation du concept d'algorithme d'approximation à temps polynomial. Le premier algorithme de ce genre est souvent attribué à Graham [129], et c'est le sujet du problème 35.5.

Depuis ces premiers travaux, on a inventé des milliers d'algorithmes d'approximation pour une foule de problèmes, et ce domaine a suscité une littérature abondante. Des ouvrages récents de Ausiello et al. [25], Hochbaum [149] et Vazirani [305] sont consacrés exclusivement aux algorithmes d'approximation, ce qui est aussi le cas d'études dues à Shmoys [277] et à Klein et Young [181]. Plusieurs autres ouvrages, tels Garey et Johnson [110] et Papadimitriou et Steiglitz [237], traitent aussi en détail des algorithmes d'approximation. Lawler, Lenstra, Rinnooy Kan et Shmoys [197] étudient en détail les algorithmes d'approximation pour le problème du voyageur de commerce.

Papadimitriou et Steiglitz attribuent l'algorithme COUVERTURE-SOMMET-APPROCHÉE à F. Gavril et M. Yannakakis. Le problème de la couverture de sommets a été étudié de façon intensive (Hochbaum [149] cite 16 algorithmes d'approximation différents pour ce problème), mais tous les ratios d'approximation sont au moins de $2 - o(1)$.

L'algorithme TOURNÉE-PRC-APPROCHÉE apparaît dans un article de Rosenkrantz, Stearns et Lewis [261]. Christofides a amélioré cet algorithme et donné un algorithme d'approximation $3/2$ pour le problème du voyageur de commerce avec inégalité triangulaire. Arora [21] et Mitchell [223] ont montré que, si les points sont dans le plan euclidien, il existe un schéma d'approximation à temps polynomial. On doit à Sahni et Gonzalez [264] la découverte du théorème 35.3.

L'analyse de l'algorithme glouton pour le problème de la couverture d'ensemble s'inspire de la démonstration publiée par Chvátal [61] d'un résultat plus général ; le résultat basique, tel qu'il est présenté ici, est dû à Johnson [166] et Lovász [206].

L'algorithme SOMME-SOUS-ENSEMBLE-APPROCHÉE et son analyse s'inspirent vaguement d'algorithmes d'approximation voisins, concernant le problème du sac-à-dos et celui de la somme de sous-ensemble, donnés par Ibarra et Kim [164].

L'algorithme randomisé pour la satisfaisabilité MAX-3-CNF est implicite dans les travaux de Johnson [166]. L'algorithme pour la couverture de sommets pondérée est dû à Hochbaum [148]. La section 35.4 ne fait qu'effleurer la puissance de la randomisation et de la programmation linéaire pour la création d'algorithmes d'approximation.

Le mélange de ces deux concepts donne une technique dite « arrondi randomisé », dans laquelle le problème est d'abord formulé en tant que programme linéaire entier. Il y a ensuite résolution de la relaxation de programmation linéaire, puis interprétation des variables de la solution en tant que probabilités. Ces probabilités servent alors à faciliter la résolution du problème originel. Cette technique a été utilisée pour la première fois par Raghavan et Thompson [255], et depuis lors elle a resservi à maintes reprises. (Voir Motwani, Naor et Raghavan [227] pour une présentation.) Entre autres concepts récents majeurs concernant les algorithmes d'approximation, citons : la méthode primal-dual (voir [116] pour une présentation) ; la recherche de coupes peu denses pour utilisation dans des algorithmes diviser-pour-régner [199] ; l'emploi de la programmation semi définie [115].

Comme mentionné dans les notes de fin du chapitre 34, des résultats récents en matière de démonstrations vérifiables probabilistiquement ont donné des minorants concernant l'approximabilité de nombreux problèmes, dont plusieurs problèmes présentés dans ce chapitre. Outre les références citées à la fin de ce chapitre, le chapitre de Arora et Lund [22] contient une bonne description des relations entre les démonstrations vérifiables probabilistiquement et la complexité de l'approximation de divers problèmes.

PARTIE 8

ANNEXES : ÉLÉMENTS DE MATHÉMATIQUES

L'analyse des algorithmes demande un certain nombre de connaissances mathématiques. Certains des outils nécessaires ne sont pas plus compliqués que l'algèbre étudiée au lycée, mais d'autres outils sont moins familiers. Dans la partie 1, vous avez vu comment manipuler des notations asymptotiques et résoudre des récurrences. Les annexes qui suivent regroupent un certain nombre d'autres concepts et méthodes utilisés pour l'analyse des algorithmes. Comme signalé dans l'introduction de la partie 1, il se pourrait que vous connaissiez déjà une bonne partie des notions contenues dans ces annexes (bien que les convention notationnelles que nous employons puissent, à l'occasion, différer de celles que vous avez vues dans d'autres ouvrages). Considérez donc ces annexes comme une partie du genre « références ». Nous y avons, cependant, inclus des exercices et des problèmes qui vous permettront d'améliorer vos connaissances théoriques.

L'annexe A présente des méthodes pour l'évaluation et l'encadrement des sommes, qui reviennent souvent dans l'analyse des algorithmes. Nombre des formules de ce chapitre figurent dans tous les bons manuels de calcul numérique, mais il est pratique de les avoir toutes en un même endroit.

L'annexe B contient des définitions et notations fondamentales concernant les ensembles, les relations, les fonctions, les graphes et les arbres. Ce chapitre donne aussi quelques propriétés basiques de ces objets mathématiques.

L'annexe C commence par les principes élémentaires du dénombrement : permutations, combinaisons, et tout le reste. Le reste du chapitre contient des définitions et des propriétés en matière de théorie élémentaire des probabilités. La plupart des algorithmes présentés dans ce livre n'exigent pas de connaissances en probabilités pour leur analyse, ce qui fait que vous pouvez facilement sauter, en première lecture, la fin du chapitre. Si vous rencontrez ultérieurement une analyse probabiliste que vous voulez mieux comprendre, servez-vous de l'annexe C comme matériel de référence.

Annexe A

Sommations

Quand un algorithme contient une structure de contrôle itératif comme une boucle **tant que** ou **pour**, on peut exprimer son temps d'exécution comme la somme des temps dépensés à chaque exécution du corps de la boucle. On a vu par exemple, dans la section 2.2, que la j -ème itération du tri par insertion prenait un temps proportionnel à j dans le pire des cas. En additionnant le temps passé sur chaque itération, nous avons obtenu la sommation (ou série) :

$$\sum_{j=2}^n j.$$

L'évaluation de cette sommation avait abouti à une borne $\Theta(n^2)$ pour le temps d'exécution de l'algorithme dans le pire des cas. Cet exemple illustre combien il est important de bien comprendre comment manipuler et borner les sommations.

La section A.1 donne plusieurs formules fondamentales concernant les sommations. La section A.2 livre des techniques utiles pour borner les sommations. Les formules de la section A.1 sont données sans preuve, quoique des démonstrations soient présentées dans la section A.2 pour illustrer certaines méthodes. La plupart des autres démonstrations peuvent être trouvées dans n'importe quel ouvrage de calcul.

A.1 FORMULES ET PROPRIÉTÉS DES SOMMATIONS

Soit une séquence de nombres a_1, a_2, \dots ; la somme finie $a_1 + a_2 + \dots + a_n$, n étant un entier non négatif, peut s'écrire

$$\sum_{k=1}^n a_k .$$

Si $n = 0$, la valeur de la sommation est 0 par définition. La valeur d'une série finie est toujours bien définie, et ses termes peuvent être additionnés dans n'importe quel ordre.

Étant donnée une séquence de nombres a_1, a_2, \dots , la somme infinie $a_1 + a_2 + \dots$ peut s'écrire

$$\sum_{k=1}^{\infty} a_k ,$$

ce qu'on interprète par

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n a_k .$$

Si la limite n'existe pas, la série **diverge**; sinon, elle **converge**. Les termes d'une série convergente ne peuvent pas toujours être additionnés dans n'importe quel ordre. On peut cependant réarranger les termes d'une **série absolument convergente**, c'est-à-dire une série $\sum_{k=1}^{\infty} a_k$ pour laquelle la série $\sum_{k=1}^{\infty} |a_k|$ converge également.

a) Linéarité

Pour tout nombre réel c et pour toute paire de suites a_1, a_2, \dots, a_n et b_1, b_2, \dots, b_n on a :

$$\sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k .$$

La propriété de linéarité est également respectée par une série infinie convergente.

La propriété de linéarité peut être utilisée pour manipuler des sommations qui contiennent des notations asymptotiques. Par exemple,

$$\sum_{k=1}^n \Theta(f(k)) = \Theta \left(\sum_{k=1}^n f(k) \right) .$$

Dans cette équation, la notation Θ dans le membre gauche s'applique à la variable k , mais dans le membre droit, elle s'applique à n . Ce type de manipulation peut aussi s'appliquer aux séries infinies convergentes.

b) Série arithmétique

La sommation

$$\sum_{k=1}^n k = 1 + 2 + \cdots + n,$$

est une *série arithmétique* et a pour valeur

$$\sum_{k=1}^n k = \frac{1}{2}n(n+1) \quad (\text{A.1})$$

$$= \Theta(n^2). \quad (\text{A.2})$$

c) Sommes des carrés et des cubes

On a les sommes suivantes de carrés et de cubes :

$$\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6}, \quad (\text{A.3})$$

$$\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}. \quad (\text{A.4})$$

d) Série géométrique

Pour un réel $x \neq 1$, la sommation

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \cdots + x^n$$

est une *série exponentielle* ou *géométrique* et a pour valeur

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}. \quad (\text{A.5})$$

Lorsque la sommation est infinie et que $|x| < 1$, on a la série géométrique décroissante

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}. \quad (\text{A.6})$$

e) Série harmonique

Pour un entier positif n , le n -ième *nombre harmonique* est

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^n \frac{1}{k} \\ &= \ln n + O(1). \end{aligned} \quad (\text{A.7})$$

(Nous prouverons cette borne à la section A.2.)

f) *Intégration et dérivation de séries*

D'autres formules peuvent être obtenues en intégrant ou dérivant les formules précédentes. Par exemple, en dérivant les deux membres de la série géométrique (A.6) et en les multipliant par x , on obtient

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} . \quad (\text{A.8})$$

pour $|x| < 1$.

g) *Séries emboîtées*

Pour une séquence a_0, a_1, \dots, a_n quelconque,

$$\sum_{k=1}^n (a_k - a_{k-1}) = a_n - a_0 , \quad (\text{A.9})$$

car chaque terme de a_1, a_2, \dots, a_{n-1} est ajouté exactement une fois et soustrait exactement une fois. On dit que la somme est **emboîtée**. De même,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n .$$

Comme exemple de somme emboîtée, considérons la série

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} .$$

Puisqu'on peut réécrire chaque terme sous la forme

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1} ,$$

on obtient

$$\begin{aligned} \sum_{k=1}^{n-1} \frac{1}{k(k+1)} &= \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1} \right) \\ &= 1 - \frac{1}{n} . \end{aligned}$$

h) *Produits*

Le produit fini $a_1 a_2 \cdots a_n$ peut s'écrire

$$\prod_{k=1}^n a_k .$$

Si $n = 0$, le produit vaut 1 par définition. On peut convertir une formule contenant un produit en une formule contenant une somme, en utilisant l'identité

$$\lg \left(\prod_{k=1}^n a_k \right) = \sum_{k=1}^n \lg a_k .$$

Exercices

A.1.1 Trouver une formule simple pour $\sum_{k=1}^n (2k - 1)$.

A.1.2 * Montrer que $\sum_{k=1}^n 1/(2k - 1) = \ln(\sqrt{n}) + O(1)$ en manipulant la série harmonique.

A.1.3 Montrer que $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ pour $0 < |x| < 1$.

A.1.4 * Montrer que $\sum_{k=0}^{\infty} (k-1)/2^k = 0$.

A.1.5 * évaluer la somme $\sum_{k=1}^{\infty} (2k+1)x^{2k}$.

A.1.6 Prouver que $\sum_{k=1}^n O(f_k(n)) = O(\sum_{k=1}^n f_k(n))$ en utilisant la linéarité des sommations.

A.1.7 évaluer le produit $\prod_{k=1}^n 2 \cdot 4^k$.

A.1.8 * évaluer le produit $\prod_{k=2}^n (1 - 1/k^2)$.

A.2 BORNES DES SOMMATIONS

Il existe de nombreuses techniques pour borner les sommations qui décrivent les temps d'exécution des algorithmes. Voici quelques-unes des méthodes les plus fréquemment utilisées.

a) Récurrence mathématique

Le moyen le plus direct pour évaluer une série est l'emploi de la récurrence mathématique. Par exemple, démontrons que la série arithmétique $\sum_{k=1}^n k$ a pour valeur $\frac{1}{2}n(n+1)$. On peut facilement le vérifier pour $n = 1$, ce qui nous permet de formuler l'hypothèse de récurrence selon laquelle l'affirmation est valable pour n , et ensuite prouver qu'elle l'est toujours pour $n+1$. On a

$$\begin{aligned} \sum_{k=1}^{n+1} k &= \sum_{k=1}^n k + (n+1) \\ &= \frac{1}{2}n(n+1) + (n+1) \\ &= \frac{1}{2}(n+1)(n+2) . \end{aligned}$$

Il n'est pas nécessaire de connaître la valeur d'une sommation pour utiliser la récurrence mathématique. On peut également s'en servir pour montrer l'existence d'une borne. Par exemple, démontrons que la série géométrique $\sum_{k=0}^n 3^k$ est en $O(3^n)$. Plus précisément, démontrons que $\sum_{k=0}^n 3^k \leq c3^n$ pour une certaine constante c . Pour la condition initiale $n = 0$, on a $\sum_{k=0}^0 3^k = 1 \leq c \cdot 1$ tant que $c \geq 1$. En supposant que cette borne est valable pour n , démontrons qu'elle l'est encore pour $n + 1$. On a :

$$\begin{aligned} \sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^n 3^k + 3^{n+1} \\ &\leq c3^n + 3^{n+1} \\ &= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \\ &\leq c3^{n+1} \end{aligned}$$

tant que $(1/3 + 1/c) \leq 1$ ou, si l'on préfère, $c \geq 3/2$. Donc $\sum_{k=0}^n 3^k = O(3^n)$, ce que nous voulions démontrer.

Il faut être prudent lorsqu'on utilise la notation asymptotique pour trouver des bornes par récurrence. Considérons la démonstration (erronée) suivante de $\sum_{k=1}^n k = O(n)$. Évidemment, $\sum_{k=1}^1 k = O(1)$. En supposant que la borne est valable pour n , on démontre maintenant sa validité pour $n + 1$:

$$\begin{aligned} \sum_{k=1}^{n+1} k &= \sum_{k=1}^n k + (n+1) \\ &= O(n) + (n+1) \quad \Leftarrow \text{faux!!} \\ &= O(n+1) . \end{aligned}$$

L'erreur dans cet argument est que la « constante » masquée par le « grand O » croît avec n et n'est donc pas constante. Nous n'avons pas montré que la constante était valable pour *tout* n .

b) Bornes des termes

Parfois, on peut obtenir un bon majorant pour une série en bornant chaque terme ; et il suffit souvent d'utiliser le terme le plus grand pour majorer les autres. Par exemple, un majorant immédiat pour la série arithmétique (A.1) est

$$\begin{aligned} \sum_{k=1}^n k &\leq \sum_{k=1}^n n \\ &= n^2 . \end{aligned}$$

En général, pour une série $\sum_{k=1}^n a_k$, si l'on prend $a_{\max} = \max_{1 \leq k \leq n} a_k$, alors

$$\sum_{k=1}^n a_k \leq na_{\max} .$$

La technique consistant à majorer chaque terme d'une série par le terme le plus grand est une méthode faible quand la série peut en fait être bornée par une série géométrique. Soit la série $\sum_{k=0}^n a_k$, supposons que $a_{k+1}/a_k \leq r$ pour tout $k \geq 0$, où $0 < r < 1$ est une constante. La somme peut être bornée par une série géométrique infinie décroissante, puisque $a_k \leq a_0 r^k$, et donc

$$\begin{aligned} \sum_{k=0}^n a_k &\leq \sum_{k=0}^{\infty} a_0 r^k \\ &= a_0 \sum_{k=0}^{\infty} r^k \\ &= a_0 \frac{1}{1-r}. \end{aligned}$$

On peut appliquer cette méthode pour borner la sommation $\sum_{k=1}^{\infty} (k/3^k)$. Pour pouvoir commencer la sommation à $k=0$, on la réécrit sous la forme $\sum_{k=0}^{\infty} ((k+1)/3^{k+1})$. Le premier terme (a_0) est $1/3$, et le rapport (r) de termes consécutifs est

$$\begin{aligned} \frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} &= \frac{1}{3} \cdot \frac{k+2}{k+1} \\ &\leq \frac{2}{3} \end{aligned}$$

pour tout $k \geq 0$. On a donc

$$\begin{aligned} \sum_{k=1}^{\infty} \frac{k}{3^k} &= \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \\ &\leq \frac{1}{3} \cdot \frac{1}{1-2/3} \\ &= 1. \end{aligned}$$

Une erreur courante lors de l'application de cette méthode consiste à montrer que le rapport de deux termes consécutifs est inférieur à 1 et à en déduire que la sommation est bornée par une série géométrique. Une illustration en est donnée par la série harmonique qui diverge, puisque

$$\begin{aligned} \sum_{k=1}^{\infty} \frac{1}{k} &= \lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{1}{k} \\ &= \lim_{n \rightarrow \infty} \Theta(\lg n) \\ &= \infty. \end{aligned}$$

Le rapport du $(k+1)$ -ème et du k -ème terme de cette série vaut $k/(k+1) < 1$, mais la série n'est pas bornée par une série géométrique décroissante. Pour borner une série par une série géométrique, on doit montrer qu'il existe un $r < 1$ constant, tel que

le rapport de toutes les paires de termes consécutifs n'excède jamais r . Dans la série harmonique, un tel r n'existe pas, car le rapport devient arbitrairement proche de 1.

c) Découpage des sommations

On peut obtenir des bornes pour une sommation difficile en exprimant la série comme la somme de deux ou plusieurs séries, en découpant l'intervalle de l'indice et en bornant chacune des séries résultantes. Par exemple, supposons qu'on essaye de trouver un minorant pour la série arithmétique $\sum_{k=1}^n k$, dont on a déjà montré qu'elle était majorée par n^2 . On pourrait essayer de minorer chaque terme de la sommation par le plus petit terme ; mais, comme ce terme vaut 1, on obtient un minorant de n , donc très éloigné du majorant n^2 .

Un meilleur minorant peut être obtenu si l'on commence par découper la sommation. Supposons pour notre confort que n soit pair. On a

$$\begin{aligned} \sum_{k=1}^n k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^n k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^n (n/2) \\ &= (n/2)^2 \\ &= \Omega(n^2), \end{aligned}$$

qui est une borne asymptotiquement approchée, puisque $\sum_{k=1}^n k = O(n^2)$.

Quand il s'agit d'une sommation apparaissant lors de l'analyse d'un algorithme, on peut souvent la découper en plusieurs morceaux, et ignorer un nombre constant de termes initiaux. En général, cette technique s'applique lorsque chaque terme a_k d'une sommation $\sum_{k=0}^n a_k$ est indépendant de n . Alors, pour une constante $k_0 > 0$ quelconque, on peut écrire

$$\begin{aligned} \sum_{k=0}^n a_k &= \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^n a_k \\ &= \Theta(1) + \sum_{k=k_0}^n a_k, \end{aligned}$$

car les termes initiaux de la sommation sont tous constants et il y en a un nombre constant. On peut alors employer d'autres méthodes pour borner $\sum_{k=k_0}^n a_k$. Cette technique s'applique aussi aux sommes infinies. Ainsi, pour trouver un majorant asymptotique pour

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k},$$

on observe que le rapport de termes consécutifs est

$$\begin{aligned} \frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+1)^2}{2k^2} \\ &\leq \frac{8}{9} \end{aligned}$$

si $k \geq 3$. La somme peut donc être découpée en

$$\begin{aligned} \sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=0}^2 \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ &\leq \sum_{k=0}^2 \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k \\ &= O(1), \end{aligned}$$

puisque la première somme a un nombre constant de termes et que la seconde est une série géométrique décroissante.

La technique du découpage des sommations peut être utilisée pour déterminer des bornes asymptotiques dans des situations beaucoup plus délicates. Par exemple, on peut obtenir une borne de $O(\lg n)$ pour la série harmonique (A.7) :

$$H_n = \sum_{k=1}^n \frac{1}{k}.$$

Le principe est de découper l'intervalle 1 à n en $\lfloor \lg n \rfloor$ parties et de majorer la contribution de chaque partie par 1. Donc,

$$\begin{aligned} \sum_{k=1}^n \frac{1}{k} &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i+j} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\ &\leq \lg n + 1. \end{aligned} \tag{A.10}$$

d) Approximation par des intégrales

Lorsqu'une sommation peut être exprimée par $\sum_{k=m}^n f(k)$, où $f(k)$ est une fonction monotone croissante, on peut l'approximer à l'aide d'intégrales :

$$\int_{m-1}^n f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_m^{n+1} f(x) dx. \tag{A.11}$$

Cette approximation est justifiée à la figure A.1. La sommation est représentée comme l'aire des rectangles de la figure, et l'intégrale est la région ombrée située sous la courbe. Lorsque $f(k)$ est une fonction monotone décroissante, on peut utiliser une méthode similaire pour fournir les bornes

$$\int_m^{n+1} f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x) dx. \tag{A.12}$$

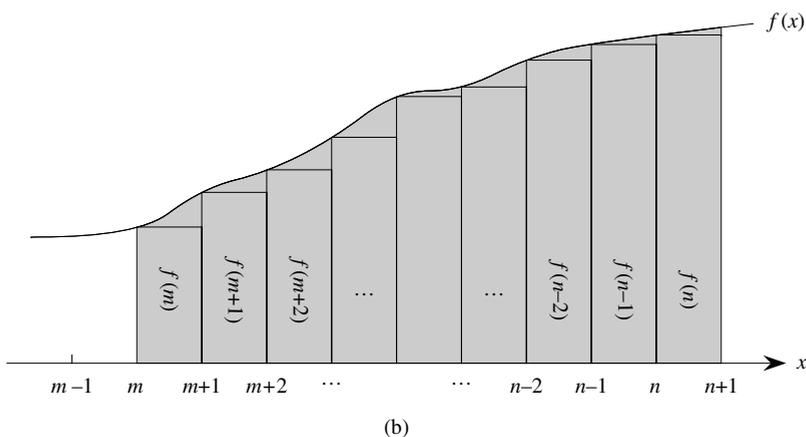
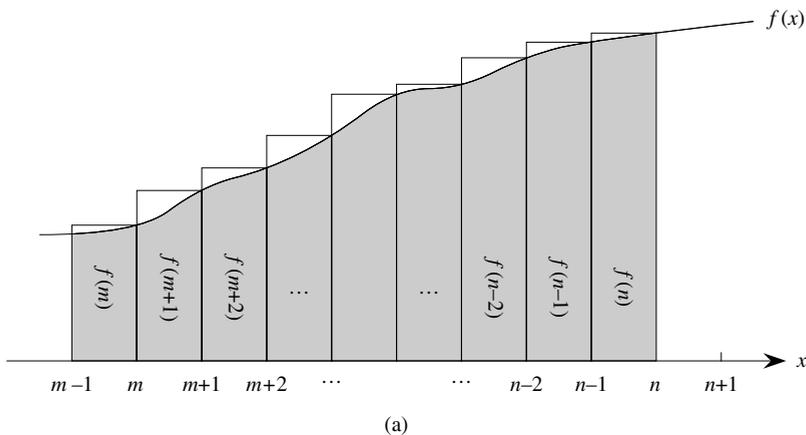


Figure A.1 Approximation de $\sum_{k=m}^n f(k)$ par des intégrales. L'aire de chaque rectangle est affichée à l'intérieur du rectangle, et la somme de ces aires représente la valeur de la sommation. L'aire de l'intégrale est représentée par la zone en gris au-dessous de la courbe. En comparant les aires en (a), on obtient $\int_{m-1}^n f(x) dx \leq \sum_{k=m}^n f(k)$, et en décalant ensuite les rectangles d'une unité vers la droite, on obtient $\sum_{k=m}^n f(k) \leq \int_m^{n+1} f(x) dx$ en (b).

L'approximation par intégrale (A.12) donne une estimation serrée du n -ième nombre harmonique. Pour un minorant, on obtient

$$\begin{aligned} \sum_{k=1}^n \frac{1}{k} &\geq \int_1^{n+1} \frac{dx}{x} \\ &= \ln(n+1). \end{aligned} \tag{A.13}$$

Pour avoir un majorant, on dérive l'inégalité

$$\begin{aligned} \sum_{k=2}^n \frac{1}{k} &\leq \int_1^n \frac{dx}{x} \\ &= \ln n, \end{aligned}$$

ce qui donne la borne

$$\sum_{k=1}^n \frac{1}{k} \leq \ln n + 1. \tag{A.14}$$

Exercices

A.2.1 Montrer que $\sum_{k=1}^n 1/k^2$ est majorée par une constante.

A.2.2 Trouver un majorant asymptotique pour la sommation

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil.$$

A.2.3 Montrer, via découpage de la sommation, que le n -ième nombre harmonique est $\Omega(\lg n)$.

A.2.4 Donner une approximation par intégrale de $\sum_{k=1}^n k^3$.

A.2.5 Pourquoi n'utilise-t-on pas l'approximation par intégrale (A.12) directement sur $\sum_{k=1}^n 1/k$ pour obtenir un majorant du n -ième nombre harmonique ?

PROBLÈMES

A.1. Bornes des sommations

Donner des bornes asymptotiquement approchées pour les sommations suivantes. On supposera que $r \geq 0$ et $s \geq 0$ sont des constantes.

$$a. \sum_{k=1}^n k^r.$$

$$b. \sum_{k=1}^n \lg^s k.$$

$$c. \sum_{k=1}^n k^r \lg^s k.$$

NOTES

Knuth [182] est une excellente référence pour les sujets abordés dans ce chapitre. Les propriétés fondamentales des suites et séries peuvent être trouvées dans de nombreux ouvrages, comme (en français) Guessarian et Arnold [325] et Froidevaux, Gaudel et Soria [324] ou (en américain) Apostol [18] et Thomas et Finney [296].

Annexe B

Ensembles, etc.

Maints chapitres de ce livre s'appuient sur des notions de mathématiques discrètes. Ce chapitre reprend de manière plus complète les notations, définitions et propriétés simples des ensembles, relations, fonctions, graphes et arbres. Les lecteurs déjà bien au fait de ces choses pourront se contenter de feuilleter ce chapitre.

B.1 ENSEMBLES

Un *ensemble* est une collection d'objets distinguables, appelés *membres* ou *éléments*. Si un objet x est un membre d'un ensemble S , on écrit $x \in S$ (lire « x est membre de S » ou, plus brièvement, « x appartient à S »). Si x n'est pas membre de S , on écrit $x \notin S$. On peut décrire un ensemble en énumérant explicitement chacun de ses membres, dans une liste entourée d'accolades. Par exemple, on peut définir un ensemble S contenant exactement les nombres 1, 2 et 3 en écrivant $S = \{1, 2, 3\}$. Comme 2 est un membre de l'ensemble S , on peut écrire $2 \in S$ et comme 4 n'est pas membre de l'ensemble, on a $4 \notin S$. Un ensemble ne peut pas contenir le même objet plus d'une fois ⁽¹⁾, et ses éléments ne sont pas ordonnés. Deux ensembles A et B sont *égaux*, ce qui s'écrit $A = B$, s'ils contiennent les mêmes éléments. Par exemple, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Des notations spéciales sont adoptées pour les ensembles le plus fréquemment rencontrés.

(1) Il existe une variante, dite *multi ensemble*, qui peut contenir un même objet plus d'une fois.

- \emptyset représente l'**ensemble vide**, c'est-à-dire l'ensemble ne contenant aucun élément.
- \mathbf{Z} représente l'ensemble des **entiers**, c'est-à-dire l'ensemble $\{\dots, -2, -1, 0, 1, 2, \dots\}$.
- \mathbf{R} représente l'ensemble des **nombres réels**.
- \mathbf{N} représente l'ensemble des **entiers naturels**, c'est-à-dire l'ensemble $\{0, 1, 2, \dots\}$ ⁽²⁾.

Si tous les éléments d'un ensemble A sont contenus dans un ensemble B , autrement dit si $x \in A$ implique $x \in B$, on écrit $A \subseteq B$ et on dit que A est un **sous-ensemble** de B . Un ensemble A est un **sous-ensemble propre** de B , noté $A \subset B$, si $A \subseteq B$ mais $A \neq B$. (Certains auteurs utilisent le symbole « \subset » pour représenter la relation de sous-ensemble ordinaire, plutôt que la relation de sous-ensemble propre.) Pour un ensemble A quelconque, on a $A \subseteq A$. Pour deux ensembles A et B , on a $A = B$ si et seulement si $A \subseteq B$ et $B \subseteq A$. Pour trois ensembles A , B et C quelconques, si $A \subseteq B$ et $B \subseteq C$, alors $A \subseteq C$. Pour un ensemble A quelconque, on a $\emptyset \subseteq A$.

On définit parfois les ensembles en fonction d'autres ensembles. Étant donné un ensemble A , on peut définir un ensemble $B \subseteq A$ en établissant une propriété qui distingue les éléments de B . Par exemple, on peut définir l'ensemble des entiers pairs par $\{x : x \in \mathbf{Z} \text{ et } x/2 \text{ est un entier}\}$. Le deux-points qui apparaît dans cette notation se lit « tel que ». (Certains auteurs utilisent une barre verticale au lieu du deux-points.)

Étant donnés deux ensembles A et B , on peut aussi définir de nouveaux ensembles en leur appliquant des **opérations ensemblistes** :

- L'**intersection** des ensembles A et B est l'ensemble

$$A \cap B = \{x : x \in A \text{ et } x \in B\} .$$

- L'**union** des ensembles A et B est l'ensemble

$$A \cup B = \{x : x \in A \text{ ou } x \in B\} .$$

- La **différence** entre deux ensembles A et B est l'ensemble

$$A - B = \{x : x \in A \text{ et } x \notin B\} .$$

Les opérations ensemblistes respectent les lois suivantes :

Lois pour l'ensemble vide :

$$\begin{aligned} A \cap \emptyset &= \emptyset , \\ A \cup \emptyset &= A . \end{aligned}$$

Lois d'idempotence :

$$\begin{aligned} A \cap A &= A , \\ A \cup A &= A . \end{aligned}$$

(2) Certains auteurs font commencer les entiers naturels à 1 et non à 0. La tendance actuelle semble être de commencer à 0.

Lois commutatives :

$$\begin{aligned}A \cap B &= B \cap A, \\A \cup B &= B \cup A.\end{aligned}$$

Lois associatives :

$$\begin{aligned}A \cap (B \cap C) &= (A \cap B) \cap C, \\A \cup (B \cup C) &= (A \cup B) \cup C.\end{aligned}$$

Lois de distributivité :

$$\begin{aligned}A \cap (B \cup C) &= (A \cap B) \cup (A \cap C), \\A \cup (B \cap C) &= (A \cup B) \cap (A \cup C).\end{aligned}\tag{B.1}$$

Lois d'absorption :

$$\begin{aligned}A \cap (A \cup B) &= A, \\A \cup (A \cap B) &= A.\end{aligned}$$

Lois de DeMorgan :

$$\begin{aligned}A - (B \cap C) &= (A - B) \cup (A - C), \\A - (B \cup C) &= (A - B) \cap (A - C).\end{aligned}\tag{B.2}$$

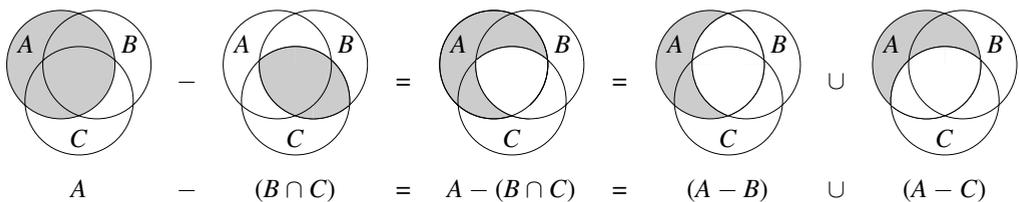


Figure B.1 Diagramme de Venn illustrant la première loi de DeMorgan (voir (B.2)). Chaque ensemble A , B et C est représenté par un cercle.

La première des lois de DeMorgan est illustrée à la figure B.1, grâce à un **diagramme de Venn**, qui est une représentation graphique dans laquelle les ensembles sont représentés par des régions du plan.

Souvent, tous les ensembles considérés sont des sous-ensembles d'un ensemble plus grand U appelé **univers**. Par exemple, si l'on considère différents ensembles formés uniquement d'entiers, un univers pertinent sera l'ensemble \mathbf{Z} des entiers. Étant donné un univers U , on définit le **complément** d'un ensemble A par $\bar{A} = U - A$. Pour

un ensemble $A \subseteq U$ quelconque, on a les lois suivantes :

$$\begin{aligned}\overline{\overline{A}} &= A, \\ A \cap \overline{A} &= \emptyset, \\ A \cup \overline{A} &= U.\end{aligned}$$

Les lois de DeMorgan (B.2) peuvent être réécrites avec les compléments. Pour deux ensembles $A, B \subseteq U$ quelconques, on a :

$$\begin{aligned}\overline{A \cap B} &= \overline{A} \cup \overline{B}, \\ \overline{A \cup B} &= \overline{A} \cap \overline{B}.\end{aligned}$$

Deux ensembles A et B sont **disjoints** s'ils n'ont aucun élément en commun, autrement dit si $A \cap B = \emptyset$. Une collection $\mathcal{S} = \{S_i\}$ d'ensembles non vides forme une **partition** d'un ensemble S si

- les ensembles sont **disjoints deux à deux**, c'est-à-dire $S_i, S_j \in \mathcal{S}$ et $i \neq j$ implique $S_i \cap S_j = \emptyset$ et
- leur union est S , c'est-à-dire

$$S = \bigcup_{S_i \in \mathcal{S}} S_i.$$

Autrement dit, \mathcal{S} forme une partition de S si chaque élément de S appartient à un seul $S_i \in \mathcal{S}$.

Le nombre d'éléments d'un ensemble est appelé le **cardinal** (ou **taille**) de l'ensemble, noté $|S|$. Deux ensembles ont le même cardinal si leurs éléments peuvent être mis en correspondance un à un. Le cardinal de l'ensemble vide est $|\emptyset| = 0$. Si le cardinal d'un ensemble est un entier naturel, on dit que l'ensemble est **fini** ; sinon, il est **infini**. Un ensemble infini dont on peut mettre les éléments en correspondance un à un avec les entiers naturels de l'ensemble \mathbf{N} est **infini dénombrable** ; sinon, il est dit **non dénombrable**. L'ensemble \mathbf{Z} des entiers est dénombrable, mais l'ensemble \mathbf{R} des réels est non dénombrable.

Pour deux ensembles finis A et B quelconques, on a l'identité

$$|A \cup B| = |A| + |B| - |A \cap B|, \quad (\text{B.3})$$

d'où l'on peut conclure que

$$|A \cup B| \leq |A| + |B|.$$

Si A et B sont disjoints, alors $|A \cap B| = 0$ et donc $|A \cup B| = |A| + |B|$. Si $A \subseteq B$, alors $|A| \leq |B|$.

Un ensemble fini de n éléments est parfois appelé un **n -uplet**. Un 1-uplet s'appelle un **singleton**. Un sous-ensemble de k éléments d'un ensemble est parfois appelé **k -sous-ensemble**.

L'ensemble de tous les sous-ensembles d'un ensemble S , ensemble vide et ensemble S compris, est noté $\mathcal{P}(S)$ et est appelé **ensemble des parties** de S . Par exemple,

$\mathcal{P}(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. L'ensemble des parties d'un ensemble fini S a pour cardinal $2^{|S|}$.

On s'intéresse parfois à des structures de type ensemble dans lesquelles les éléments sont ordonnés. Un **couple** (ou paire ordonnée) formé par deux éléments a et b est noté (a, b) et peut être défini formellement comme l'ensemble $(a, b) = \{a, \{a, b\}\}$. Le couple (a, b) n'est donc *pas* identique au couple (b, a) .

Le **produit cartésien** de deux ensembles A et B , noté $A \times B$, est l'ensemble de tous les couples tels que le premier élément soit élément de A et le second soit élément de B . De manière plus formelle,

$$A \times B = \{(a, b) : a \in A \text{ et } b \in B\} .$$

Par exemple, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Lorsque A et B sont des ensembles finis, le cardinal de leur produit cartésien est :

$$|A \times B| = |A| \cdot |B| . \quad (\text{B.4})$$

Le produit cartésien de n ensembles A_1, A_2, \dots, A_n est l'ensemble des **n -uplets**

$$A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) : a_i \in A_i, i = 1, 2, \dots, n\} ,$$

dont le cardinal vaut

$$|A_1 \times A_2 \times \dots \times A_n| = |A_1| \cdot |A_2| \cdot \dots \cdot |A_n|$$

si tous les ensembles sont finis. On représente le produit cartésien appliqué n fois successivement sur un ensemble A par l'ensemble

$$A^n = \underbrace{A \times A \times \dots \times A}_{n \text{ fois}} .$$

dont le cardinal est $|A^n| = |A|^n$ si A est fini. Un n -uplet peut également être vu comme une séquence finie de longueur n (voir page 1041).

Exercices

B.1.1 Illustrer la première loi de distributivité (B.1) à l'aide de diagrammes de Venn.

B.1.2 Démontrer que les lois de DeMorgan peuvent être généralisées à une collection quelconque d'ensemble finis :

$$\begin{aligned} \overline{A_1 \cap A_2 \cap \dots \cap A_n} &= \overline{A_1} \cup \overline{A_2} \cup \dots \cup \overline{A_n} , \\ \overline{A_1 \cup A_2 \cup \dots \cup A_n} &= \overline{A_1} \cap \overline{A_2} \cap \dots \cap \overline{A_n} . \end{aligned}$$

B.1.3 * Démontrer qu'on peut généraliser l'équation (B.3), appelée *principe d'inclusion et d'exclusion* :

$$\begin{aligned}
 |A_1 \cup A_2 \cup \dots \cup A_n| = & \\
 & |A_1| + |A_2| + \dots + |A_n| \\
 & - |A_1 \cap A_2| - |A_1 \cap A_3| - \dots \quad (\text{toutes les paires}) \\
 & + |A_1 \cap A_2 \cap A_3| + \dots \quad (\text{tous les triplets}) \\
 & \vdots \\
 & + (-1)^{n-1} |A_1 \cap A_2 \cap \dots \cap A_n|.
 \end{aligned}$$

B.1.4 Montrer que l'ensemble des entiers naturels impairs est dénombrable.

B.1.5 Montrer que, pour tout ensemble fini S , l'ensemble des parties $\mathcal{P}(S)$ a $2^{|S|}$ éléments (c'est-à-dire qu'il existe $2^{|S|}$ sous-ensembles distincts de S).

B.1.6 Donner une définition inductive (récurrence) d'un n -uplet en étendant la définition ensembliste d'un couple.

B.2 RELATIONS

Une *relation binaire* R sur deux ensembles A et B est un sous-ensemble du produit cartésien $A \times B$. Si $(a, b) \in R$, on écrit parfois $a R b$. Quand on dit que R est une relation binaire sur un ensemble A , cela signifie que R est un sous-ensemble de $A \times A$. Par exemple, la relation « inférieur à » sur les entiers naturels est l'ensemble $\{(a, b) : a, b \in \mathbf{N} \text{ et } a < b\}$. Une relation n -aire sur les ensembles A_1, A_2, \dots, A_n est un sous-ensemble de $A_1 \times A_2 \times \dots \times A_n$.

Une relation binaire $R \subseteq A \times A$ est *réflexive* si

$$a R a$$

pour tout $a \in A$. Par exemple, « = » et « ≤ » sont des relations réflexives sur \mathbf{N} , mais « < » n'en est pas une. La relation R est *symétrique* si

$$a R b \text{ implique } b R a$$

pour tout $a, b \in A$. Par exemple, « = » est symétrique, mais « < » et « ≤ » ne le sont pas. La relation R est dite *transitive* si

$$a R b \text{ et } b R c \text{ implique } a R c$$

quels que soient $a, b, c \in A$. Par exemple, les relations « < », « ≤ », et « = » sont transitives, mais la relation $R = \{(a, b) : a, b \in \mathbf{N} \text{ et } a = b - 1\}$ ne l'est pas, puisque $3 R 4$ et $4 R 5$ n'impliquent pas $3 R 5$.

Une relation à la fois réflexive, symétrique et transitive est une *relation d'équivalence*. Par exemple, « = » est une relation d'équivalence sur les entiers naturels, mais

« $<$ » n'en est pas une. Si R est une relation d'équivalence sur un ensemble A , alors pour $a \in A$, la **classe d'équivalence** de a est l'ensemble $[a] = \{b \in A : a R b\}$, c'est-à-dire l'ensemble de tous les éléments équivalents à a . Par exemple, si l'on définit $R = \{(a, b) : a, b \in \mathbf{N} \text{ et } a + b \text{ est un nombre pair}\}$, alors R est une relation d'équivalence, puisque $a + a$ est pair (réflexive), $a + b$ pair implique $b + a$ pair (symétrique) et $a + b$ pair et $b + c$ pair implique $a + c$ pair (transitive). La classe d'équivalence de 4 est $[4] = \{0, 2, 4, 6, \dots\}$ et la classe d'équivalence de 3 est $[3] = \{1, 3, 5, 7, \dots\}$.

Voici un théorème fondamental sur les classes d'équivalence :

Théorème B.1 (Une relation d'équivalence est identique à une partition) *Les classes d'équivalence d'une relation d'équivalence R quelconque sur un ensemble A forment une partition de A , et toute partition de A détermine une relation d'équivalence sur A dont les classes d'équivalence sont les ensembles de la partition.*

Démonstration : Pour la première partie de la preuve, on doit montrer que les classes d'équivalence de R sont des ensembles non vides, disjoints deux à deux et dont l'union est A . Comme R est réflexive, $a \in [a]$, donc les classes d'équivalence ne sont pas vides ; par ailleurs, puisque chaque élément $a \in A$ appartient à la classe d'équivalence $[a]$, l'union des classes d'équivalence est A . Il reste à montrer que les classes d'équivalence sont deux à deux disjointes, c'est-à-dire que si deux classes d'équivalence $[a]$ et $[b]$ ont un élément c en commun, elles forment en réalité un seul et même ensemble. Or $a R c$ et $b R c$, ce qui, d'après la symétrie et la transitivité, implique $a R b$. Ainsi, pour un élément arbitraire $x \in [a]$, on a $x R a$ implique $x R b$ et donc $[a] \subseteq [b]$. De la même manière, $[b] \subseteq [a]$ et donc $[a] = [b]$.

Pour la seconde partie de la démonstration, soit $\mathcal{A} = \{A_i\}$ une partition de A et soit $R = \{(a, b) : \text{il existe } i \text{ tel que } a \in A_i \text{ et } b \in A_i\}$. Nous disons que R est une relation d'équivalence sur A . La relation est réflexive, puisque $a \in A_i$ implique $a R a$. Idem pour la symétrie, car si $a R b$, alors a et b sont dans le même ensemble A_i et donc $b R a$. Si $a R b$ et $b R c$, alors les trois éléments sont tous dans le même ensemble et donc $a R c$, ce qui valide la transitivité. Pour voir que les ensembles de la partition sont les classes d'équivalence de R , observez que, si $a \in A_i$, alors $x \in [a]$ implique $x \in A_i$ et $x \in A_i$ implique $x \in [a]$. \square

Une relation binaire R sur un ensemble A est **antisymétrique** si

$$a R b \text{ et } b R a \text{ implique } a = b .$$

Par exemple, la relation « \leq » sur les entiers naturels est antisymétrique, puisque $a \leq b$ et $b \leq a$ implique $a = b$. Une relation réflexive, antisymétrique et transitive est une **relation d'ordre partiel** et un ensemble sur lequel est définie une relation d'ordre partiel est appelé **ensemble partiellement ordonné**. Par exemple, la relation « est un descendant de » est une relation d'ordre partiel sur l'ensemble de tous les êtres humains (si l'on considère que chaque individu est son propre descendant).

Dans un ensemble partiellement ordonné A , il se peut qu'il n'y ait aucun élément x « maximal » unique tel que $y R x$ pour tout $y \in A$. En revanche, il peut exister plusieurs éléments **maximaux** x tels que, pour aucun $y \in A$, on n'ait $x R y$. Par exemple,

dans une collection de boîtes de tailles différentes, il peut y avoir plusieurs boîtes maximales, qui ne tiennent dans aucune autre boîte, sans qu'il existe pour autant de boîte « maximum » unique dans laquelle n'importe quelle autre boîte pourrait tenir.

Une relation d'ordre partiel R sur un ensemble A définit un ordre **total** ou **ordre linéaire** si, pour tout $a, b \in A$, on a aRb ou bRa , autrement dit, si chaque paire d'éléments de A peut être reliée par R . Par exemple, la relation « \leq » est un ordre total sur les entiers naturels, mais la relation « est un descendant de » n'est pas un ordre total sur tous les êtres humains, puisqu'on peut trouver deux individus dont aucun n'est descendant de l'autre.

Exercices

B.2.1 Démontrer que la relation \subseteq , « est sous-ensemble de », sur tous les sous-ensembles de \mathbf{Z} est un ordre partiel, mais pas un ordre total.

B.2.2 Montrer que, pour tout entier positif n , la relation « équivalent modulo n » est une relation d'équivalence sur les entiers. (On dit que $a \equiv b \pmod{n}$ s'il existe un entier q tel que $a - b = qn$.) Dans quelles classes d'équivalence cette relation partitionne-t-elle les entiers ?

B.2.3 Donner des exemples de relations qui sont

- réflexives et symétriques, mais pas transitives,
- réflexives et transitives, mais pas symétriques,
- symétriques et transitives, mais pas réflexives.

B.2.4 Soit S un ensemble fini et soit R une relation d'équivalence sur $S \times S$. Montrer que, si en outre R est antisymétrique, alors les classes d'équivalence de S définies par R sont des singletons.

B.2.5 Le Professeur Narcisse affirme que, si une relation R est symétrique et transitive, elle est également réflexive. Il propose la preuve suivante : d'après la symétrie, aRb implique bRa . La transitivité implique donc que aRa . Le professeur a-t-il raison ?

B.3 FONCTIONS

Étant donnés deux ensembles A et B , une **fonction** f est une relation binaire sur $A \times B$ telle que pour tout $a \in A$, il existe un $b \in B$ et un seul tel que $(a, b) \in f$. L'ensemble A est appelé **domaine de définition** de f et l'ensemble B est appelé **domaine de valeurs** de f . On écrit parfois $f : A \rightarrow B$; si $(a, b) \in f$, on écrit $b = f(a)$, puisque b est déterminé de manière unique par le choix de a .

Intuitivement, la fonction f assigne un élément de B à chaque élément de A . Aucun élément de A n'est relié à deux éléments différents de B , mais le même élément de B peut être assigné à deux éléments différents de A . Par exemple, la relation binaire

$$f = \{(a, b) : a \in \mathbf{N} \text{ et } b = a \bmod 2\}$$

est une fonction $f : \mathbf{N} \rightarrow \{0, 1\}$, puisque pour chaque entier naturel a , il existe une valeur b de $\{0, 1\}$ et une seule, telle que $b = a \bmod 2$. Pour cet exemple, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ etc. À l'inverse, la relation binaire

$$g = \{(a, b) : a \in \mathbf{N} \text{ et } a + b \text{ est pair}\}$$

n'est pas une fonction, puisque $(1, 3)$ et $(1, 5)$ sont tous les deux dans g et donc, si l'on prend $a = 1$, il n'existe pas de b unique tel que $(a, b) \in g$.

Étant donnée une fonction $f : A \rightarrow B$, si $b = f(a)$, on dit que a est l'**argument** de f et que b est la **valeur** de f en a . On peut définir une fonction en donnant sa valeur pour chaque élément de son domaine de définition. Par exemple, on pourrait définir $f(n) = 2n$ pour $n \in \mathbf{N}$, autrement dit $f = \{(n, 2n) : n \in \mathbf{N}\}$. Deux fonctions f et g sont dites **égales** si elles ont les mêmes domaines de définition et de valeurs et si, pour tout a du domaine de définition, $f(a) = g(a)$.

Une **suite finie** de longueur n est une fonction f dont le domaine de définition est l'ensemble $\{0, 1, \dots, n-1\}$. On définit souvent une suite finie en donnant la liste de ses valeurs : $\langle f(0), f(1), \dots, f(n-1) \rangle$. Une **suite infinie** est une fonction dont le domaine de définition est l'ensemble \mathbf{N} des entiers naturels. Par exemple, la suite de Fibonacci, définie par (3.21), est la suite infinie $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \dots \rangle$.

Quand le domaine de définition d'une fonction f est un produit cartésien, on omet souvent les parenthèses supplémentaires qui encadrent l'argument de f . Par exemple, si $f : A_1 \times A_2 \times \dots \times A_n \rightarrow B$, on écrira $b = f(a_1, a_2, \dots, a_n)$ au lieu de $b = f((a_1, a_2, \dots, a_n))$. On dit aussi que chaque a_i est un **argument** de la fonction f , bien qu'en toute rigueur l'argument (unique) de f soit le n -uplet (a_1, a_2, \dots, a_n) .

Si $f : A \rightarrow B$ est une fonction et $b = f(a)$, on dit parfois que b est l'**image** de a par f . L'image d'un ensemble $A' \subseteq A$ par f est défini par

$$f(A') = \{b \in B : b = f(a) \text{ pour un certain } a \in A'\} .$$

L'**ensemble des valeurs** de f , souvent notée $Im(f)$, est l'image de son domaine de définition, autrement dit $f(A)$. Par exemple, l'ensemble des valeurs de la fonction $f : \mathbf{N} \rightarrow \mathbf{N}$ définie par $f(n) = 2n$ est $f(\mathbf{N}) = \{m : m = 2n \text{ pour un certain } n \in \mathbf{N}\}$.

Une fonction est une **surjection** si l'ensemble de ses valeurs est égal à son domaine de valeurs. Par exemple, la fonction $f(n) = \lfloor n/2 \rfloor$ est une fonction surjective de \mathbf{N} vers \mathbf{N} , car chaque élément de \mathbf{N} est une valeur de f pour un certain argument. Par contre, la fonction $f(n) = 2n$ n'est pas une fonction surjective de \mathbf{N} vers \mathbf{N} , car aucun argument de f n'est capable de produire la valeur 3. En revanche, la fonction $f(n) = 2n$ est une fonction surjective de l'ensemble des entiers naturels vers l'ensemble des entiers pairs. On dit parfois qu'une surjection $f : A \rightarrow B$ établit une correspondance de A sur B .

Une fonction $f : A \rightarrow B$ est une **injection** si des arguments distincts de f produisent des valeurs distinctes, autrement dit si $a \neq a'$ implique $f(a) \neq f(a')$. Par exemple, la fonction $f(n) = 2n$ est une fonction injective de \mathbf{N} vers \mathbf{N} , puisque chaque nombre pair b est l'image par f d'au plus un élément du domaine de définition, dans ce cas $b/2$. La fonction $f(n) = \lfloor n/2 \rfloor$ n'est pas injective, puisque la valeur 1 peut être produite par deux arguments : 2 et 3.

Une fonction $f : A \rightarrow B$ est une **bijection** si elle est à la fois injective et surjective. Par exemple, la fonction $f(n) = (-1)^n \lfloor n/2 \rfloor$ est une bijection de \mathbf{N} vers \mathbf{Z} :

$$\begin{aligned} 0 &\rightarrow 0, \\ 1 &\rightarrow -1, \\ 2 &\rightarrow 1, \\ 3 &\rightarrow -2, \\ 4 &\rightarrow 2, \\ &\vdots \end{aligned}$$

La fonction est injective, car aucun élément de \mathbf{Z} n'est image de plus d'un élément de \mathbf{N} . Elle est surjective, puisque chaque élément de \mathbf{Z} est image d'un certain élément de \mathbf{N} . Donc, la fonction est bijective. Une bijection d'un ensemble A vers lui-même est parfois appelée **permutation**.

Quand une fonction f est bijective, son **inverse** f^{-1} est définie par

$$f^{-1}(b) = a \text{ si et seulement si } f(a) = b.$$

Par exemple, l'inverse de la fonction $f(n) = (-1)^n \lfloor n/2 \rfloor$ est

$$f^{-1}(m) = \begin{cases} 2m & \text{si } m \geq 0, \\ -2m - 1 & \text{si } m < 0. \end{cases}$$

Exercices

B.3.1 Soient A et B des ensembles finis et soit $f : A \rightarrow B$ une fonction. Montrer que

- a. si f est injective, alors $|A| \leq |B|$;
- b. si f est surjective, alors $|A| \geq |B|$.

B.3.2 La fonction $f(x) = x+1$ est-elle bijective quand les domaines de définition et de valeurs sont \mathbf{N} ? Et quand ils sont \mathbf{Z} ?

B.3.3 Donner une définition naturelle de l'inverse d'une relation binaire telle que, si une relation est en fait une fonction bijective, son inverse relationnel soit égal à son inverse fonctionnel.

B.3.4 * Donner une bijection de \mathbf{Z} vers $\mathbf{Z} \times \mathbf{Z}$.

B.4 GRAPHES

Cette section présente deux sortes de graphes : orientés et non orientés. Le lecteur doit avoir à l'esprit que certaines définitions de la littérature diffèrent de celles données ici, bien que pour la plupart, les différences soient minimales. La section 22.1 montre comment les graphes peuvent être représentés dans la mémoire d'un ordinateur.

Un **graphe orienté** G est représenté par un couple (S, A) , où S est un ensemble fini et A est une relation binaire sur S . L'ensemble S est appelé **ensemble des sommets** de G . On dit que l'ensemble A est l'**ensemble des arcs** de G . La figure B.2(a) est une représentation graphique d'un graphe orienté dont l'ensemble des sommets est $\{1, 2, 3, 4, 5, 6\}$. Les sommets sont représentés par des cercles sur la figure et les arcs sont représentés par des flèches. Notez que les **boucles**, c'est-à-dire les arcs reliant un sommet à lui-même, sont possibles.

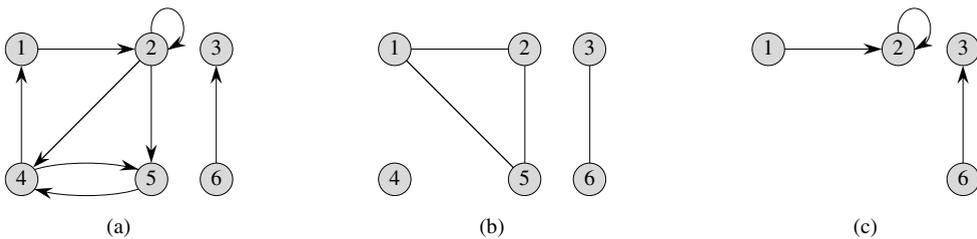


Figure B.2 Graphes orientés et non orientés. (a) Un graphe orienté $G = (S, A)$, où $S = \{1, 2, 3, 4, 5, 6\}$ et $A = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. L'arc $(2, 2)$ est une boucle. (b) Un graphe non orienté $G = (S, A)$, où $S = \{1, 2, 3, 4, 5, 6\}$ et $A = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. Le sommet 4 est isolé. (c) Le sous-graphe du graphe de la partie (a) induit par l'ensemble de sommets $\{1, 2, 3, 6\}$.

Dans un **graphe non orienté** $G = (S, A)$, l'ensemble des **arêtes** A n'est pas constitué de couples, mais de paires de sommets *non ordonnées*. Autrement dit, une arête est un ensemble $\{u, v\}$, où $u, v \in S$ et $u \neq v$. Par convention, on utilise la notation (u, v) pour représenter une arête, de préférence à la notation $\{u, v\}$ et l'on considère que (u, v) et (v, u) représentent la même arête. Dans un graphe non orienté, les boucles sont interdites et chaque arête est donc exactement constituée de deux sommets distincts. La figure B.2(b) est une représentation graphique d'un graphe non orienté sur l'ensemble de sommets $\{1, 2, 3, 4, 5, 6\}$.

Beaucoup de définitions sont identiques pour les graphes orientés et non orientés, mais certains termes ont des significations différentes selon qu'ils sont employés dans l'un ou l'autre contexte. Si (u, v) est un arc d'un graphe orienté $G = (V, E)$, on dit que (u, v) est **incident extérieurement**, ou qu'il **part**, du sommet u et qu'il est **incident intérieurement**, ou qu'il **arrive**, au sommet v . Ainsi, les arcs partant du sommet 2 sur

la figure B.2(a) sont $(2, 2)$, $(2, 4)$ et $(2, 5)$. Les arcs arrivant au sommet 2 sont $(1, 2)$ et $(2, 2)$. Si (u, v) est une arête d'un graphe non orienté $G = (V, E)$, on dit que (u, v) est **incidente** pour les sommets u et v . Sur la figure B.2(b), les arêtes incidentes pour le sommet 2 sont $(1, 2)$ et $(2, 5)$.

Si (u, v) est un arc d'un graphe $G = (S, A)$, on dit que le sommet v est **adjacent** au sommet u . Quand le graphe est non orienté, la relation d'adjacence est symétrique. Quand le graphe est orienté, la relation d'adjacence n'est pas forcément symétrique. Si v est adjacent à u dans un graphe orienté, on écrit parfois $u \rightarrow v$. Dans les parties (a) et (b) de la figure B.2, le sommet 2 est adjacent au sommet 1, puisque l'arc $(1, 2)$ appartient aux deux graphes. Le sommet 1 n'est *pas* adjacent au sommet 2 dans la figure B.2(a), puisque l'arc $(2, 1)$ n'appartient pas au graphe.

Le **degré** d'un sommet dans un graphe non orienté est le nombre d'arêtes qui lui sont incidentes. Par exemple, le sommet 2 de la figure B.2(b) a pour degré 2. Un sommet dont le degré est 0, comme le sommet 4 de la figure B.2(b), est **isolé**. Dans un graphe orienté, le **degré sortant** d'un sommet est le nombre d'arcs qui en partent et le **degré entrant** d'un sommet est le nombre d'arcs qui y arrivent. Le **degré** d'un sommet dans un graphe orienté est égal à la somme de son degré entrant et de son degré sortant. Le sommet 2 de la figure B.2(a) a pour degré entrant 2, pour degré sortant 3 et pour degré 5.

Un **chemin de longueur** k d'un sommet u vers un sommet u' dans un graphe orienté $G = (S, A)$ est une suite $\langle v_0, v_1, v_2, \dots, v_k \rangle$ de sommets tels que $u = v_0$, $u' = v_k$ et $(v_{i-1}, v_i) \in A$ pour $i = 1, 2, \dots, k$. La longueur du chemin est le nombre d'arcs dans le chemin. Le chemin **contient** les sommets v_0, v_1, \dots, v_k et les arcs $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. (Il existe toujours un chemin de longueur 0 entre u et u .) S'il existe un chemin p de u à u' , on dit que u' est **accessible** depuis u via p , ce qu'on écrit parfois $u \rightsquigarrow u'$. Un chemin est **élémentaire** si tous les sommets du chemin sont distincts. Dans la figure B.2(a), le chemin $\langle 1, 2, 5, 4 \rangle$ est un chemin simple de longueur 3. Le chemin $\langle 2, 5, 4, 5 \rangle$ n'est pas simple.

Un **sous-chemin** d'un chemin $p = \langle v_0, v_1, \dots, v_k \rangle$ est une sous-suite contiguë de ses sommets. Autrement dit, pour des $0 \leq i \leq j \leq k$ quelconques, la sous-suite de sommets $\langle v_i, v_{i+1}, \dots, v_j \rangle$ est un sous-chemin de p .

Dans un graphe orienté, un chemin $\langle v_0, v_1, \dots, v_k \rangle$ forme un **circuit** si $v_0 = v_k$ et si le chemin contient au moins un arc. Ce circuit est **élémentaire** si, en plus, v_1, v_2, \dots, v_k sont distincts. Une boucle est un circuit de longueur 1. Deux chemins $\langle v_0, v_1, v_2, \dots, v_{k-1}, v_0 \rangle$ et $\langle v'_0, v'_1, v'_2, \dots, v'_{k-1}, v'_0 \rangle$ forment le même circuit s'il existe un entier j tel que $v'_i = v_{(i+j) \bmod k}$ pour $i = 0, 1, \dots, k - 1$. Dans la figure B.2(a), le chemin $\langle 1, 2, 4, 1 \rangle$ forme le même circuit que les chemins $\langle 2, 4, 1, 2 \rangle$ et $\langle 4, 1, 2, 4 \rangle$. Ce circuit est élémentaire, contrairement au circuit $\langle 1, 2, 4, 5, 4, 1 \rangle$. Le circuit $\langle 2, 2 \rangle$ formé par l'arc $(2, 2)$ est une boucle. On dit d'un graphe orienté sans boucles qu'il est **simple**. Dans un graphe non orienté, une chaîne $\langle v_0, v_1, \dots, v_k \rangle$ forme un **cycle** (**élémentaire**) si $k \geq 3$, $v_0 = v_k$ et v_1, v_2, \dots, v_k sont distincts. Par exemple, dans la figure B.2(b), le chemin $\langle 1, 2, 5, 1 \rangle$ est un cycle. Un graphe sans cycle est dit **acyclique**.

Un graphe non orienté est **connexe** si chaque paire de sommets est reliée par une chaîne. Les **composantes connexes** d'un graphe sont constituées des classes d'équivalence de sommets induites par la relation « est accessible depuis ». Le graphe de la figure B.2(b) comprend trois composantes connexes : $\{1, 2, 5\}$, $\{3, 6\}$ et $\{4\}$. Chaque sommet de $\{1, 2, 5\}$ est accessible à partir de n'importe quel autre sommet de $\{1, 2, 5\}$. Un graphe non orienté est connexe s'il comporte une composante connexe et une seule, autrement dit, si chaque sommet est accessible depuis tous les autres sommets.

Un graphe orienté est **fortement connexe** si, pour tout couple de sommets, chacun est accessible depuis l'autre. Les **composantes fortement connexes** d'un graphe forment les classes d'équivalence de la relation entre sommets « sont accessibles mutuellement ». Un graphe orienté est fortement connexe s'il a une seule composante fortement connexe. Le graphe de la figure B.2(a) comprend trois composantes fortement connexes : $\{1, 2, 4, 5\}$, $\{3\}$ et $\{6\}$. Toutes les paires de sommets de $\{1, 2, 4, 5\}$ sont mutuellement accessibles. Les sommets $\{3, 6\}$ ne forment pas une composante fortement connexe, car le sommet 6 ne peut pas être atteint à partir du sommet 3.

Deux graphes $G = (S, A)$ et $G' = (S', A')$ sont dits **isomorphes** s'il existe un bijection $f : S \rightarrow S'$ telle que $(u, v) \in A$ si et seulement si $(f(u), f(v)) \in A'$. Autrement dit, on peut rebaptiser les sommets de G avec les étiquettes des sommets de G' sans modifier les arcs correspondants dans G et G' . La figure B.3(a) montre deux graphes isomorphes G et G' ayant pour ensembles de sommets respectifs $S = \{1, 2, 3, 4, 5, 6\}$ et $S' = \{u, v, w, x, y, z\}$. La correspondance entre V et V' donnée par $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$ est la fonction bijective requise. Les graphes de la figure B.3(b) ne sont pas isomorphes. Bien que les deux graphes aient 5 sommets et 7 arêtes, le graphe supérieur comporte un sommet de degré 4, contrairement au graphe inférieur.

On dit qu'un graphe $G' = (S', A')$ est un **sous-graphe** de $G = (S, A)$ si $S' \subseteq S$ et $A' \subseteq A$. Étant donné un ensemble $S' \subseteq S$, le sous-graphe de G **engendré** par S' est le graphe $G' = (S', A')$, où

$$A' = \{(u, v) \in A : u, v \in S'\} .$$

Le sous-graphe engendré par l'ensemble de sommets $\{1, 2, 3, 6\}$ dans la figure B.2(a) apparaît dans la figure B.2(c) et a pour ensemble d'arcs $\{(1, 2), (2, 2), (6, 3)\}$.

Étant donné un graphe non orienté $G = (S, A)$, la **version orientée** de G est le graphe orienté $G' = (S, A')$, où $(u, v) \in A'$ si et seulement si $(u, v) \in A$. Autrement dit, chaque arête (u, v) de G est remplacée dans la version orientée par les deux arcs (u, v) et (v, u) . Étant donné un graphe orienté $G = (S, A)$, la **version non orientée** de G est le graphe non orienté $G' = (S, A')$, où $(u, v) \in A'$ si et seulement si $u \neq v$ et $(u, v) \in A$. Autrement dit, la version non orientée contient les arcs de G après « suppression des orientations » et élimination des boucles. (Comme (u, v) et (v, u) représentent une seule et même arête dans un graphe non orienté, la version non orientée d'un graphe orienté ne la contient qu'une seule fois, même si le graphe orienté contient les deux

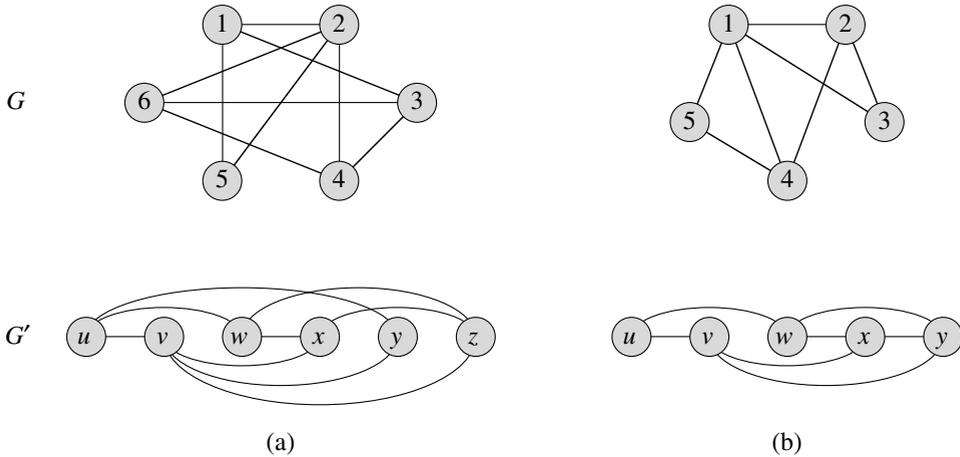


Figure B.3 (a) Deux graphes isomorphes. Les sommets du graphe supérieur peuvent être reliés un à un à ceux du graphe inférieur par $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$. (b) Ces deux graphes ne sont pas isomorphes, puisque le graphe supérieur comporte un sommet de degré 4, contrairement au graphe inférieur.

arcs (u, v) et (v, u) .) Dans un graphe orienté $G = (S, A)$, le **voisin** d'un sommet u est un sommet quelconque adjacent à u dans la version non orientée de G . Autrement dit, v est un voisin de u si $u \neq v$ et si l'on a $(u, v) \in A$ ou $(v, u) \in A$. Dans un graphe non orienté, u et v sont voisins s'ils sont adjacents.

Certains types de graphes portent des noms particuliers. Un **graphe complet** est un graphe non orienté dans lequel les sommets sont tous adjacents deux à deux. Un graphe **biparti** est un graphe non orienté $G = (S, A)$ dans lequel S peut être partitionné en deux ensembles S_1 et S_2 tels que $(u, v) \in A$ implique soit $u \in S_1$ et $v \in S_2$, soit $u \in S_2$ et $v \in S_1$. En d'autres termes, toutes les arêtes passent entre les deux ensembles S_1 et S_2 . Un graphe non orienté acyclique est une **forêt** et un graphe non orienté connexe acyclique est un **arbre** (voir Section B.5).

Il existe deux variantes de graphes que vous pourrez rencontrer à l'occasion. Un **multigraphe** est une sorte de graphe non orienté, mais qui peut comporter des arêtes multiples entre les sommets ainsi que des boucles. Un **hypergraphe** est une sorte de graphe non orienté, mais chaque **hyperarête**, au lieu de relier deux sommets, relie un sous-ensemble arbitraire de sommets. De nombreux algorithmes écrits pour des graphes orientés ou non orientés peuvent être adaptés pour s'exécuter sur ce type de structures.

La **contraction** d'un graphe non orienté $G = (V, E)$ selon une arête $e = (u, v)$ est un graphe $G' = (V', E')$, où $V' = V - \{u, v\} \cup \{x\}$ et x est un nouveau sommet. L'ensemble des arêtes E' est formé à partir de E , via les opérations suivantes : suppression de l'arête (u, v) ; pour tout sommet w incident à u ou à v , suppression de celle des arêtes (u, w) et (v, w) qui est dans E ; ajout de la nouvelle arête (x, w) .

Exercices

B.4.1 Au cours d'une soirée, les convives se serrent la main les uns les autres et chacun se souvient du nombre de fois qu'il ou elle a serré des mains. À la fin de la soirée, le maître de cérémonie additionne le nombre de fois que chaque invité a serré une main. Montrer que le résultat est pair en démontrant le **lemme de la poignée de main** : si $G = (S, A)$ est un graphe non orienté, alors

$$\sum_{v \in S} \text{degré}(v) = 2 |A| .$$

B.4.2 Montrer que, si un graphe orienté ou non contient un chemin entre deux sommets u et v , alors il contient un chemin élémentaire entre u et v . Montrer que, si un graphe orienté contient un circuit, alors il contient un circuit élémentaire.

B.4.3 Montrer qu'un graphe quelconque $G = (S, A)$ non orienté et connexe satisfait $|A| \geq |S| - 1$.

B.4.4 Vérifier que, dans un graphe non orienté, la relation « est accessible depuis » est une relation d'équivalence pour les sommets d'un graphe. Parmi les trois propriétés d'une relation d'équivalence, quelles sont celles qui restent vraies pour la relation « est accessible depuis » sur les sommets d'un graphe orienté ?

B.4.5 Quelle est la version non orientée du graphe orienté de la figure B.2(a) ? Quelle est la version orientée du graphe non orienté de la figure B.2(b) ?

B.4.6 * Montrer qu'un hypergraphe peut être représenté par un graphe biparti si l'on fait correspondre l'incidence dans l'hypergraphe à l'adjacence dans le graphe biparti. (*Conseil* : Faire correspondre un ensemble de sommets du graphe biparti aux sommets de l'hypergraphe, et faire correspondre l'autre ensemble de sommets du graphe biparti aux hyperarêtes.)

B.5 ARBRES

Comme pour les graphes, les arbres peuvent se définir de différentes façons, légèrement différentes entre elles. Cette section donne des définitions et des propriétés mathématiques pour plusieurs types d'arbre. Les sections 10.4 et 22.1 décrivent comment les arbres peuvent être représentés dans une mémoire d'ordinateur.

B.5.1 Arbres libres

Comme nous l'avons dit dans la section B.4, un **arbre** est un graphe non orienté, connexe et acyclique. Si un graphe non orienté est acyclique mais pas forcément connexe, on dit que c'est une **forêt**. Nombre d'algorithmes pour les arbres sont également valables pour les forêts. La figure B.4(a) montre un arbre et la figure B.4(b) montre une forêt. La forêt de la figure B.4(b) n'est pas un arbre parce qu'elle n'est pas

connexe. Le graphe de la figure B.4(c) n'est ni un arbre ni une forêt, car il contient un cycle.

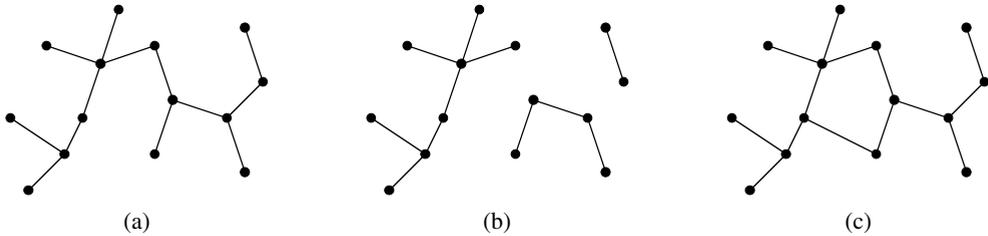


Figure B.4 (a) Un arbre. (b) Une forêt. (c) Un graphe contenant un cycle, et qui n'est donc ni un arbre ni une forêt.

Le théorème suivant englobe plusieurs faits importants concernant les arbres.

Théorème B.2 (Propriétés des arbres libres) Soit $G = (S, A)$ un graphe non orienté. Les affirmations suivantes sont équivalentes.

- 1) G est un arbre.
- 2) Deux sommets quelconques de G sont reliés par une chaîne élémentaire unique.
- 3) G est connexe mais, si l'on enlève un sommet quelconque à A , le graphe résultant n'est plus connexe.
- 4) G est connexe et $|A| = |S| - 1$.
- 5) G est acyclique et $|A| = |S| - 1$.
- 6) G est acyclique, mais si une arête quelconque est ajoutée à A , le graphe résultant contient un cycle.

Démonstration : (1) \Rightarrow (2) : Puisqu'un arbre est connexe, deux sommets quelconques de G sont reliés par au moins une chaîne élémentaire. Soient u et v deux sommets reliés par deux chaînes élémentaires distinctes p_1 et p_2 , comme illustré sur la figure B.5. Soit w le sommet à partir duquel les chaînes commencent à diverger ; en d'autres termes, w est le premier sommet situé à la fois sur p_1 et p_2 et dont le successeur sur p_1 est x et le successeur sur p_2 est y , où $x \neq y$. Soit z le premier sommet à partir duquel les chaînes convergent à nouveau ; autrement dit, z est le premier sommet après w à être à la fois sur p_1 et sur p_2 . Soit p' la sous-chaîne de p_1 qui part de w et rejoint z en passant par x et soit p'' la sous-chaîne de p_2 qui part de w et rejoint z en passant par y . Les chaînes p' et p'' n'ont aucun sommet commun, hormis leurs extrémités. Du coup, la chaîne obtenue par concaténation de p' et de l'inverse de p'' est un cycle. Cela contredit notre hypothèse que G est un arbre. Donc, si G est un arbre, il ne peut y avoir au plus qu'une chaîne élémentaire entre deux sommets.

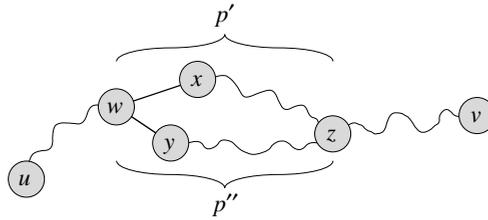


Figure B.5 Une étape dans la démonstration du théorème B.2 : si (1) G est un arbre, alors (2) deux sommets quelconques de G sont reliés par une chaîne élémentaire unique. On suppose, en raisonnant par l'absurde, que les sommets u et v sont reliés par deux chaînes élémentaires distinctes p_1 et p_2 . Ces chaînes commencent à diverger au sommet w et recommencent à converger au sommet z . La chaîne p' concaténée avec l'inverse de la chaîne p'' crée un cycle, ce qui conduit à une contradiction.

(2) \Rightarrow (3) : Si deux sommets quelconques de G sont reliés par une chaîne élémentaire unique, alors G est connexe. Soit (u, v) une arête quelconque de A . Cette arête est une chaîne reliant u à v ; elle doit donc être la seule chaîne entre u et v . Si l'on supprime (u, v) de G , il n'existe plus aucune chaîne entre u et v et donc G n'est plus connexe.

(3) \Rightarrow (4) : Par hypothèse, le graphe G est connexe et, d'après l'exercice B.4.3, on a $|A| \geq |S| - 1$. Nous allons démontrer $|A| \leq |S| - 1$ par récurrence. Un graphe connexe à $n = 1$ ou $n = 2$ sommets a $n - 1$ arête. Supposons que G ait $n \geq 3$ sommets et que tous les graphes satisfaisant à (3) qui ont moins de n sommets satisfassent également à $|A| \leq |S| - 1$. Si l'on retire une arête arbitraire de G , on sépare le graphe en $k \geq 2$ composantes connexes (en fait $k = 2$). Chaque composante satisfait à (3), sans quoi G ne satisferait pas à (3). Donc, par récurrence, le nombre des arêtes de toutes les composantes confondues vaut au plus $|S| - k \leq |S| - 2$. Si l'on ajoute l'arête manquante supprimée, on obtient $|A| \leq |S| - 1$.

(4) \Rightarrow (5) : Supposons que G soit connexe et que $|A| = |S| - 1$. Il faut montrer que G est acyclique. On suppose que G comporte un cycle contenant k sommets v_1, v_2, \dots, v_k ; on peut supposer, sans nuire à la généralité, que ce cycle est élémentaire. Soit $G_k = (S_k, A_k)$ le sous-graphe de G constitué de ce cycle. Notez que $|S_k| = |A_k| = k$. Si $k < |S|$, il doit exister un sommet $s_{k+1} \in S - S_k$ qui est adjacent à un certain sommet $s_i \in S_k$, puisque G est connexe. On définit $G_{k+1} = (S_{k+1}, A_{k+1})$ comme étant le sous-graphe de G avec $S_{k+1} = S_k \cup \{s_{k+1}\}$ et $A_{k+1} = A_k \cup \{(s_i, s_{k+1})\}$. Notez que $|S_{k+1}| = |A_{k+1}| = k + 1$. Si $k + 1 < |S|$, on peut continuer, en définissant G_{k+2} de la même manière, etc. jusqu'à ce que l'on obtienne $G_n = (S_n, A_n)$, où $n = |S|$, $S_n = S$ et $|A_n| = |S_n| = |S|$. Comme G_n est un sous-graphe de G , on a $A_n \subseteq A$ et donc $|A| \geq |S|$, ce qui contredit l'hypothèse que $|A| = |S| - 1$. Par conséquent, G est acyclique.

(5) \Rightarrow (6) : Supposons que G soit acyclique et que $|A| = |S| - 1$. Soit k le nombre de composantes connexes de G . Chaque composante connexe est un arbre par définition et, comme (1) implique (5), la somme de toutes les arêtes de toutes les composantes connexes de G vaut $|S| - k$. Par suite, on doit avoir k égal à 1, et G est en fait un arbre. Comme (1) implique (2), deux sommets quelconques de G sont reliés par une chaîne élémentaire unique. Donc, l'ajout d'une arête quelconque à G crée un cycle. (6) \Rightarrow (1) : Supposons que G soit acyclique, mais que l'ajout d'une arête quelconque à A entraîne la création d'un cycle. On doit montrer que G est connexe. Soient u et v deux

sommets arbitraires de G . Si u et v ne sont pas déjà adjacents, l'ajout de l'arête (u, v) crée un cycle dans lequel toutes les arêtes hormis (u, v) appartiennent à G . Donc, il existe une chaîne de u vers v et, puisque u et v ont été choisis arbitrairement, G est connexe. \square

B.5.2 Arbrescences et arbrescences ordonnées

Une *arbrescence* est un arbre dans lequel l'un des sommets se distingue des autres. Ce sommet particulier s'appelle la *racine* de l'arbrescence. Un sommet d'une arbrescence enracinée est souvent appelé *nœud*⁽³⁾. La figure B.6(a) montre une arbrescence pour un ensemble de 12 nœuds, avec la racine 7.

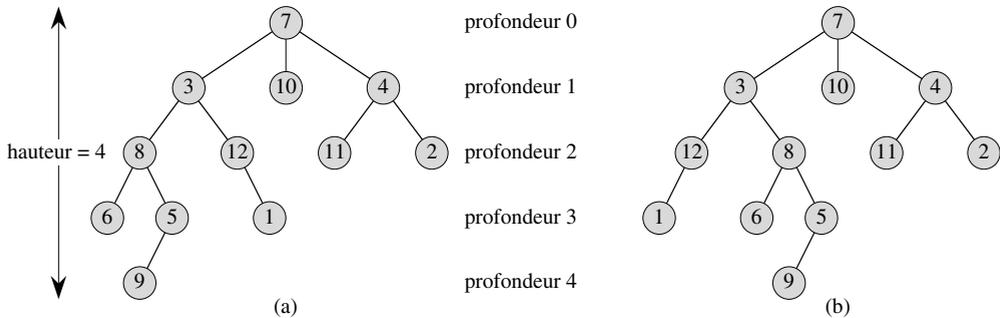


Figure B.6 Arbrescences et arbrescences ordonnées. (a) Arbre enraciné de hauteur 4. L'arbrescence est dessinée de manière classique : la racine (nœud 7) se trouve en haut, ses enfants (les nœuds de profondeur 1) sont juste en-dessous, leurs enfants (les nœuds de profondeur 2) se trouvent en-dessous et ainsi de suite. Si l'arbrescence est ordonnée, l'ordre relatif gauche-droite dans lequel sont disposés les enfants a son importance ; sinon, il n'en a pas. (b) Une autre arbrescence. En tant qu'arbrescence, elle est identique à l'arbrescence représentée en (a) ; mais en tant qu'arbrescence ordonnée, elle est différente, puisque les enfants du nœud 3 apparaissent dans un ordre différent.

On considère un nœud x d'une arbrescence T de racine r . Un nœud y quelconque sur le chemin unique allant de r à x est appelé *ancêtre* de x . Si y est un ancêtre de x , alors x est un *descendant* de y . (Chaque nœud est en même temps ancêtre et descendant de lui-même). Si y est un ancêtre de x et que $x \neq y$, alors y est un *ancêtre propre* de x et x est un *descendant propre* de y . La *sous-arbrescence de racine x* est l'arbrescence composée des descendants de x et ayant pour racine x . Par exemple, la sous-arbrescence enracinée au nœud 8 de la figure B.6(a) contient les nœuds 8, 6, 5 et 9.

Si le dernier arc du chemin menant de la racine r d'une arbrescence T vers un nœud x est (y, x) , alors y est le *parent* de x et x est un *enfant* de y . La racine est le seul nœud de T sans parent. Si deux nœuds ont le même parent, on dit qu'ils sont frères. Un nœud sans enfant est un *nœud externe* ou *feuille*. Un nœud qui n'est pas une feuille est un *nœud interne*.

(3) Le terme « nœud » est souvent utilisé dans la littérature de la théorie des graphes comme synonyme de « sommet ». Nous réserverons le terme « nœud » pour signifier un sommet d'une arbrescence.

Le nombre d'enfants d'un nœud x dans une arborescence T est appelé le *degré* de x .⁽⁴⁾ La longueur du chemin entre la racine r et un nœud x est la *profondeur* de x dans T . La *hauteur* d'un nœud dans une arborescence est le nombre d'arcs du chemin élémentaire le plus long qui relie le nœud à une feuille ; la hauteur d'une arborescence est la hauteur de sa racine. La hauteur d'une arborescence est égale à la profondeur maximale d'un quelconque nœud de cette arborescence.

Une *arborescence ordonnée* est une arborescence dans lequel les enfants de chaque nœud sont ordonnés. En d'autres termes, si un nœud a k enfants, il existe un premier enfant, un deuxième enfant, ..., et un k ième enfant. Les deux arborescences de la figure B.6 sont différentes si on les regarde comme des arborescences ordonnées, mais identiques si on les regarde comme des arborescences.

B.5.3 Arbres binaires et arborescences numérotées

Les arborescences binaires sont définies récursivement. Une *arborescence binaire* T est une structure définie sur un ensemble fini de nœuds qui :

- ne contient aucun nœud, ou qui
- est formée de trois ensembles de nœuds disjoints : un nœud *racine*, une arborescence binaire appelée *sous-arborescence de gauche* et une arborescence binaire appelée *sous-arborescence de droite*.

L'arborescence binaire qui ne contient aucun nœud est appelée *arborescence vide*, parfois noté NIL. Si la sous-arborescence de gauche n'est pas vide, sa racine est appelée *enfant de gauche* de la racine de l'arborescence entière. De même, la racine d'une sous-arborescence de droite non vide est l'*enfant de droite* de la racine de l'arborescence toute entière. Si une sous-arborescence est l'arborescence nulle NIL, on dit que l'enfant est *absent* ou *manquant*. La figure B.7(a) montre une arborescence binaire.

Une arborescence binaire n'est pas simplement une arborescence ordonnée pour laquelle chaque nœud a un degré au plus égal à 2. Par exemple, dans une arborescence binaire, si un nœud n'a qu'un seul enfant, la position de l'enfant, à savoir s'il est l'*enfant de gauche* ou l'*enfant de droite*, a son importance. Dans une arborescence ordonnée, on ne peut pas dire d'un enfant unique qu'il est de gauche ou de droite. La figure B.7(b) montre une arborescence binaire qui diffère de celui de la figure B.7(a), à cause de la position d'un nœud. Toutefois, quand on les regarde comme des arborescences ordonnées, les deux arborescences sont identiques.

Les informations de position dans une arborescence binaire peuvent être représentées par les nœuds internes d'une arborescence ordonnée, comme illustré dans la figure B.7(c). L'idée est de remplacer chaque enfant manquant dans l'arborescence binaire par un nœud sans enfants. Ces nœuds feuille sont représentés par des carrés sur la figure. L'arborescence qui en résulte est une *arborescence binaire complète* :

(4) Notez que le degré d'un nœud est différent selon qu'on considère T comme un arbre ou une arborescence. Le degré d'un sommet dans un arbre est, comme dans un graphe non orienté quelconque, le nombre de ses sommets adjacents. En revanche, dans une arborescence, le degré correspond au nombre d'enfants (le parent du nœud ne compte pas pour le calcul du degré).

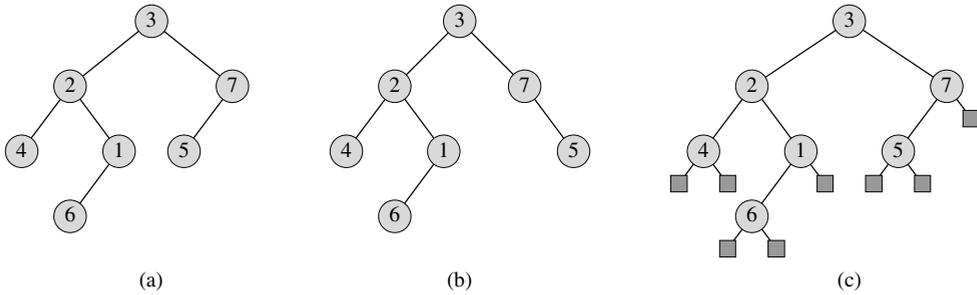


Figure B.7 Arborences binaires. (a) Arborence binaire dessinée de manière classique. L'enfant de gauche d'un nœud est placé sous le nœud et à gauche. L'enfant de droite est placé sous le nœud et à droite. (b) Arborence binaire différente de celle représentée en (a). En (a), l'enfant de gauche du nœud 7 est 5 et l'enfant de droite est absent. En (b), l'enfant de gauche du nœud 7 est absent et l'enfant de droite est 5. Si on les regarde comme des arborences ordonnées, ces arborences sont identiques, mais en tant qu'arborences binaires, elles sont distinctes. (c) L'arborence binaire en (a) représentée par les nœuds internes d'une arborence binaire complète : une arborence ordonnée dans laquelle chaque nœud interne a le degré 2. Les feuilles de l'arborence sont représentées par des carrés.

chaque nœud est soit une feuille, soit un nœud de degré égal à 2. Il n'existe aucun nœud de degré 1. Par suite, l'ordre des enfants d'un nœud a pour effet de conserver les données de position.

Les informations de position, qui distinguent les arborences binaires des arborences ordonnées, peuvent être étendues aux arborences ayant plus de 2 enfants par nœud. Dans une *arborence numérotée*, les enfants d'un nœud sont étiquetés par des entiers positifs distincts. Le i ème enfant d'un nœud est *absent* si aucun enfant n'est étiqueté par l'entier i . Un arbre k -aire est une arborence numérotée dans laquelle, pour chaque nœud, tous les enfants ayant des étiquettes supérieures à k sont manquants. Une arborence binaire est donc une arborence d'arité 2.

Une *arborence complète d'arité k* est une arborence d'arité k où toutes les feuilles ont la même profondeur et tous les nœuds internes ont le degré k . La figure B.8 montre une arborence binaire complète de hauteur 3. Combien de feuilles y a-t-il dans une arborence complète d'arité k et de hauteur h ? La racine a k enfants à la profondeur 1, chacun d'eux ayant k enfants à la profondeur 2 etc. Le nombre de feuilles à la profondeur h est donc k^h . En conséquence, la hauteur d'une arborence complète d'arité k à n feuilles est $\log_k n$. Le nombre de nœuds internes d'une arborence k -aire complète de hauteur h est :

$$\begin{aligned} 1 + k + k^2 + \dots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

d'après l'équation (A.5). Une arborence binaire complète a donc $2^h - 1$ nœuds internes.

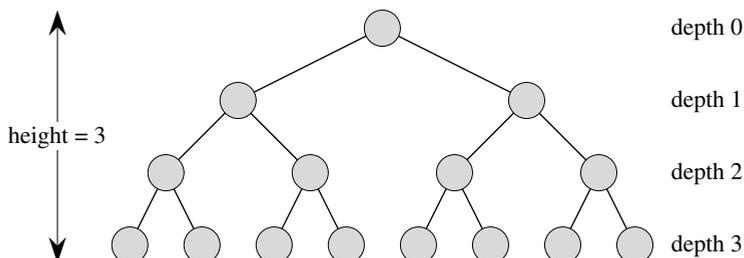


Figure B.8 Arborescence binaire complète de hauteur 3, comprenant 8 feuilles et 7 nœuds internes.

Exercices

B.5.1 Dessiner toutes les arborescences libres composées des 3 sommets A , B et C . Dessiner toutes les arborescences enracinées ayant les nœuds A , B et C et ayant A pour racine. Dessiner toutes les arborescences ordonnées ayant les nœuds A , B et C et ayant A pour racine. Dessiner toutes les arborescences binaires ayant les nœuds A , B et C et ayant A pour racine.

B.5.2 Soit $G = (S, A)$ un graphe orienté sans circuit contenant un sommet $s_0 \in S$ tel qu'il existe un chemin unique entre s_0 et chaque sommet $s \in S$. Démontrer que la version non orientée de G forme un arbre.

B.5.3 Montrer par récurrence que le nombre de nœuds de degré 2 d'une arborescence binaire non vide est inférieur d'1 au nombre de feuilles.

B.5.4 Montrer par récurrence qu'une arborescence binaire non vide à n nœuds a une hauteur d'au moins $\lfloor \lg n \rfloor$.

B.5.5 * La *longueur de chemin interne* d'une arborescence binaire complète est la somme, prise sur tous les nœuds internes de l'arborescence, de la profondeur de chaque nœud. De même, la *longueur de chemin externe* est la somme, prise sur toutes les feuilles de l'arborescence, de la profondeur de chaque feuille. On considère une arborescence binaire complète à n nœuds internes, ayant pour longueur de chemin interne i et pour longueur de chemin externe e . Démontrer que $e = i + 2n$.

B.5.6 * On associe un « poids » $w(x) = 2^{-d}$ à chaque feuille x de profondeur d d'une arborescence binaire T . Démontrer que $\sum_x w(x) \leq 1$, la somme étant calculée sur toutes les feuilles x de T . (Cette propriété est connue sous le nom d'*inégalité de Kraft*.)

B.5.7 * Montrer que, si $L \geq 2$, alors toute arborescence binaire à L feuilles contient une sous-arborescence qui a entre $L/3$ et $2L/3$ feuilles.

PROBLÈMES

B.1. Coloration de graphes

Une *k-coloration* d'un graphe non orienté $G = (S, A)$ est une fonction $c : S \rightarrow \{0, 1, \dots, k - 1\}$ telle que $c(u) \neq c(v)$ pour toute arête $(u, v) \in A$. Autrement dit, les entiers $0, 1, \dots, k - 1$ représentent les k couleurs et les sommets adjacents doivent avoir des couleurs différentes.

- a. Montrer que tout arbre est 2-coloriable.
- b. Montrer que les énoncés suivants sont équivalents :
 - 1) G est biparti.
 - 2) G est 2-coloriable.
 - 3) G n'a aucun cycle de longueur impaire.
- c. Soit d le degré maximal d'un sommet quelconque d'un graphe G . Démontrer que G peut être colorié avec $d + 1$ couleurs.
- d. Montrer que, si G a $O(|S|)$ arêtes, alors G peut être colorié avec $O(\sqrt{|S|})$ couleurs.

B.2. Graphes d'amitié

Réécrire chacun des énoncés suivants sous la forme d'un théorème pour graphes non orientés, puis en faire la démonstration. On supposera que l'amitié est symétrique, mais pas réflexive.

- a. Dans un groupe quelconque de $n \geq 2$ personnes, il existe deux personnes ayant le même nombre d'amis dans le groupe.
- b. Chaque groupe de six personnes contient soit trois amis mutuels, soit trois personnes étrangères les unes aux autres.
- c. Un groupe de personnes quelconque peut être divisé en deux sous-groupes tels que la moitié au moins des amis de chaque personne appartient au sous-groupe dont cette personne n'est *pas* membre.
- d. Si chaque personne d'un groupe est l'amie d'au moins la moitié des personnes du groupe, alors le groupe peut être réparti autour d'une table de telle façon que chacun soit assis entre deux amis.

B.3. Bisection d'arborescences

De nombreux algorithmes diviser-pour-régner opérant sur les graphes demandent que le graphe soit divisé en deux sous-graphes à peu près égaux en taille, induits *via* une partition des sommets. Ce problème va étudier les bisections d'arborescence formées *via* suppression d'un petit nombre d'arcs. Il sera impératif ici que, chaque fois que deux sommets se retrouveront dans la même sous-arborescence après suppression d'arcs, ils appartiennent à la même partition.

- a. Montrer qu'en éliminant un seul arc, il est possible de partitionner les sommets d'une arborescence binaire à n sommets en deux ensembles A et B tels que $|A| \leq 3n/4$ et $|B| \leq 3n/4$.
- b. Montrer que la constante $3/4$ de la partie (a) est optimale dans le cas le plus défavorable, en donnant un exemple d'arborescence binaire simple dont la partition la plus équilibrée induite par l'élimination d'un seul arc a pour cardinal $|A| = 3n/4$.
- c. Montrer qu'en supprimant au plus $O(\lg n)$ arcs, on peut partitionner les sommets d'une arborescence binaire à n sommets en deux ensembles A et B tels que $|A| = \lfloor n/2 \rfloor$ et $|B| = \lceil n/2 \rceil$.

NOTES

G. Boole fut à l'origine du développement de la logique symbolique et il introduisit nombre des notations ensemblistes fondamentales dans un livre publié en 1854. La théorie moderne des ensembles a été créée par G. Cantor entre 1874 et 1895. Cantor s'intéressa surtout aux ensembles de cardinal infini. Le terme « fonction » est attribué à G. W. Leibnitz, qui l'utilisa pour faire référence à plusieurs types de formules mathématiques. Sa définition limitée a été généralisée de nombreuses fois. La théorie des graphes est née en 1736, quand L. Euler démontra qu'il était impossible de traverser chacun des sept ponts de la ville de Königsberg une fois exactement pour revenir au point de départ.

Le livre de Harary [138] regroupe de nombreuses définitions et résultats de la théorie des graphes, de même que le livre de Berge [232] (en français).

Annexe C

Dénombrement et probabilités

Ce chapitre est un rappel des notions élémentaires de la théorie combinatoire et probabiliste. Si vous avez un bon bagage dans ces domaines, vous pourrez parcourir rapidement le début de ce chapitre, et ne vous concentrer que sur les dernières sections. La plupart des chapitres de ce livre ne font pas appel aux probabilités, mais pour certains, elles sont indispensables.

Dans la section C.1, nous réviserons les résultats élémentaires de la théorie du dénombrement, notamment les formules standard concernant les permutations et les combinaisons. Les axiomes de probabilités et les faits essentiels concernant les distributions probabilistes sont présentés dans la section C.2. Les variables aléatoires sont présentées dans la section C.3, en même temps que les propriétés d'espérance et de variance. La section C.4 se penche sur les distributions géométrique et binomiale qui émergent de l'étude des épreuves de Bernoulli. L'étude de la distribution binomiale est prolongée dans la section C.5, qui est une étude avancée sur les « queues » de distribution.

C.1 DÉNOMBREMENT

La théorie du dénombrement tente de répondre à la question « Combien ? » sans pour autant procéder à une énumération. Par exemple, on pourrait demander : « Combien existe-t-il de nombres différents à n bits ? » ou « De combien de manières différentes peut-on ordonner n éléments distincts ? ». Dans cette section, nous passerons en revue les éléments de la théorie du dénombrement. Une partie de ces informations étant

subordonnées à des connaissances élémentaires sur les ensembles, nous conseillons au lecteur de commencer par réviser les informations de la section B.1.

a) Règles de la somme et du produit

Un ensemble d'éléments que nous souhaitons dénombrer peut parfois être exprimé comme l'union d'ensembles disjoints, ou comme le produit cartésien de plusieurs ensembles.

La **règle de la somme** dit que le nombre de façons de choisir un élément appartenant à un ensemble parmi deux ensembles *disjoints* est la somme des cardinaux de ces deux ensembles. Autrement dit, si A et B sont deux ensembles finis sans élément commun, alors $|A \cup B| = |A| + |B|$, ce qui se déduit de l'équation (B.3). Par exemple, chaque position sur une plaque d'immatriculation de voiture est occupée par une lettre ou un chiffre. Le nombre de possibilités pour chaque position est donc $26 + 10 = 36$, puisqu'il existe 26 choix possibles si c'est une lettre et 10 choix possibles s'il s'agit d'un chiffre.

La **règle du produit** dit que le nombre de façons de choisir un couple (paire ordonnée) est le nombre de façons de choisir le premier élément, multiplié par le nombre de façons de choisir le second élément. Autrement dit, si A et B sont deux ensembles finis, alors $|A \times B| = |A| \cdot |B|$, ce qui est tout simplement l'équation (B.4). Par exemple, si un vendeur de glaces propose 28 parfums et 4 sirops différents, le nombre de *sundaes* possibles avec une boule de glace et un sirop est $28 \cdot 4 = 112$.

b) Chaînes

Une **chaîne** sur un ensemble fini E est une séquence d'éléments de E . Par exemple, il existe 8 chaînes binaires de longueur 3 :

000, 001, 010, 011, 100, 101, 110, 111 .

On appelle parfois une chaîne de longueur k une **k -chaîne**. Une **sous-chaîne** s' d'une chaîne s est une séquence ordonnée d'éléments consécutifs de s . Une **k -sous-chaîne** d'une chaîne est une sous-chaîne de longueur k . Par exemple, 010 est une 3-sous-chaîne de 01101001 (la 3-sous-chaîne qui commence à la position 4), mais 111 n'est pas une sous-chaîne de 01101001.

Une k -chaîne sur un ensemble E peut être vue comme un élément du produit cartésien E^k de k -uplets ; du coup, il existe $|E|^k$ chaînes de longueur k . Par exemple, le nombre de k -chaînes binaires est 2^k . Intuitivement, pour construire une k -chaîne sur un n -ensemble, on peut choisir le premier élément de n façons ; pour chacun de ces choix, le deuxième élément peut être choisi de n façons différentes ; et ainsi de suite, k fois. Cette construction aboutit au produit multiple $n \cdot n \cdots n = n^k$ comme nombre de k -chaînes possibles.

c) *Permutations*

Une **permutation** d'un ensemble fini E est une séquence ordonnée de tous les éléments de E , chaque élément apparaissant exactement une fois. Par exemple, si $E = \{a, b, c\}$, il existe 6 permutations de E :

$$abc, acb, bac, bca, cab, cba .$$

Il existe $n!$ permutations d'un ensemble de n éléments, puisque le premier élément de la séquence peut être choisi de n façons différentes, le deuxième de $n - 1$ façons, le troisième de $n - 2$ façons, etc.

Une **k -permutation** de E est une séquence ordonnée de k éléments de E , sans qu'un élément apparaisse plus d'une fois dans la séquence. (Une permutation ordinaire est donc simplement une n -permutation d'un n -ensemble.) Les douze 2-permutations de l'ensemble $\{a, b, c, d\}$ sont

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc .$$

Le nombre de k -permutations d'un n -ensemble est

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!} , \quad (\text{C.1})$$

puisque'il existe n façons possibles de choisir le premier élément, $n - 1$ façons de choisir le deuxième élément, etc., jusqu'à ce qu'on ait choisi k éléments, le dernier étant sélectionné parmi $n - k + 1$ éléments.

d) *Combinaisons*

Une **k -combinaison** d'un n -ensemble E est tout simplement un k -sous-ensemble de E . Il existe six 2-combinaisons du 4-ensemble $\{a, b, c, d\}$:

$$ab, ac, ad, bc, bd, cd .$$

(On utilise ici le raccourci consistant à représenter le 2-ensemble $\{a, b\}$ par ab , etc.) On peut construire une k -combinaison d'un n -ensemble en choisissant k éléments distincts (différents) dans le n -ensemble.

Le nombre de k -combinaisons d'un n -ensemble peut être exprimé en fonction du nombre de k -permutations d'un n -ensemble. Pour chaque k -combinaison, il existe exactement $k!$ permutations de ses éléments, chacune d'elle étant une k -permutation distincte du n -ensemble. Ainsi le nombre de k -combinaisons d'un n -ensemble est le nombre de k -permutations divisé par $k!$; d'après l'équation (C.1), cette quantité vaut

$$\frac{n!}{k!(n-k)!} . \quad (\text{C.2})$$

Pour $k = 0$, cette formule nous dit que le nombre de façons de choisir 0 élément dans un n -ensemble est 1 (et non 0), puisque $0! = 1$.

e) *Coefficients binomiaux*

On utilise la notation $\binom{n}{k}$ (lire « C n k ») pour représenter le nombre de combinaisons d'ordre k d'un ensemble à n éléments. D'après l'équation (C.2), on a

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Cette formule est symétrique en k et $n - k$:

$$\binom{n}{k} = \binom{n}{n-k}. \quad (\text{C.3})$$

Ces nombres sont aussi appelés **coefficients binomiaux**, à cause de leur apparition dans le **développement binomial** :

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}. \quad (\text{C.4})$$

Lorsque $x = y = 1$, on a le cas particulier :

$$2^n = \sum_{k=0}^n \binom{n}{k}.$$

Cette formule correspond au dénombrement des 2^n n -chaînes binaires en fonction du nombre de 1 qu'elles contiennent : il y a $\binom{n}{k}$ n -chaînes binaires contenant 1 exactement k fois, puisqu'il existe $\binom{n}{k}$ manières de choisir k positions parmi les n dans lesquelles on peut placer un 1.

De nombreuses identités s'appliquent aux coefficients binomiaux. Les exercices situés à la fin de cette section permettent d'en démontrer quelques-unes.

f) *Bornes binomiales*

On a parfois besoin de borner la taille d'un coefficient binomial. Pour $1 \leq k \leq n$, on a le minorant

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &= \binom{n}{k} \binom{n-1}{k-1} \cdots \binom{n-k+1}{1} \\ &\geq \left(\frac{n}{k}\right)^k. \end{aligned}$$

En s'aidant de l'inégalité $k! \geq (k/e)^k$ déduite de la formule de Stirling (3.17), on peut obtenir les majorants :

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &\leq \frac{n^k}{k!} \\ &\leq \left(\frac{en}{k}\right)^k. \end{aligned} \quad (\text{C.5})$$

Pour tout $0 \leq k \leq n$, on peut utiliser la récurrence (voir exercice C.1.12) pour démontrer l'existence du majorant

$$\binom{n}{k} \leq \frac{n^n}{k^k(n-k)^{n-k}}, \quad (\text{C.6})$$

où l'on suppose, pour des raisons pratiques, que $0^0 = 1$. Pour $k = \lambda n$, où $0 \leq \lambda \leq 1$, on peut écrire cette borne autrement :

$$\begin{aligned} \binom{n}{\lambda n} &\leq \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^\lambda \left(\frac{1}{1-\lambda}\right)^{1-\lambda} \right)^n \\ &= 2^{nH(\lambda)}, \end{aligned}$$

où

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda) \lg(1-\lambda) \quad (\text{C.7})$$

est la *fonction d'entropie (binaire)* et où, pour des raisons pratiques, on pose que $0 \lg 0 = 0$, et ainsi $H(0) = H(1) = 0$.

Exercices

C.1.1 Combien de k -sous-chaînes sont-elles contenues dans une n -chaîne ? (On considère comme différentes des k -sous-chaînes identiques situées à des positions différentes.) Combien de sous-chaînes sont-elles contenues au total dans une n -chaîne ?

C.1.2 Une fonction de $\{\text{VRAI, FAUX}\}^n$ vers $\{\text{VRAI, FAUX}\}^m$ est appelée *fonction booléenne* à n entrées et m sorties. Combien existe-t-il de fonctions booléennes à n entrées et 1 sortie ? Et combien à n entrées et m sorties ?

C.1.3 De combien de façons différentes peuvent s'asseoir n ministres autour d'une table ronde de conférence ? On considère que deux placements sont identiques si l'un peut être obtenu par rotation d'un autre.

C.1.4 De combien de manières peut-on choisir trois nombres distincts dans l'ensemble $\{1, 2, \dots, 100\}$ de façon que leur somme soit paire ?

C.1.5 Prouver l'identité

$$\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1} \quad (\text{C.8})$$

pour $0 < k \leq n$.

C.1.6 Prouver l'identité

$$\binom{n}{k} = \frac{n}{n-k} \binom{n-1}{k}$$

pour $0 \leq k < n$.

C.1.7 Pour choisir k objets parmi n , on peut distinguer l'un des objets et regarder si l'objet distingué a été choisi. Utiliser cette approche pour démontrer que

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

C.1.8 A l'aide du résultat de l'exercice C.1.7, construire un tableau pour $n = 0, 1, \dots, 6$ et $0 \leq k \leq n$ des coefficients binomiaux $\binom{n}{k}$ avec $\binom{0}{0}$ en haut, $\binom{1}{0}$ et $\binom{1}{1}$ sur la ligne suivante, et ainsi de suite. Ce type de tableau de coefficients binomiaux s'appelle *triangle de Pascal*.

C.1.9 Démontrer que

$$\sum_{i=1}^n i = \binom{n+1}{2}.$$

C.1.10 Montrer que, quels que soient $n \geq 0$ et $0 \leq k \leq n$, la valeur maximale de $\binom{n}{k}$ est atteinte lorsque $k = \lfloor n/2 \rfloor$ ou $k = \lceil n/2 \rceil$.

C.1.11 * Montrer que, quels que soient $n \geq 0, j \geq 0, k \geq 0$ et $j+k \leq n$,

$$\binom{n}{j+k} \leq \binom{n}{j} \binom{n-j}{k}. \quad (\text{C.9})$$

Donner à la fois une preuve algébrique et une argumentation fondée sur une méthode permettant de choisir $j+k$ éléments parmi n . Donner un exemple pour lequel l'égalité ne tient pas.

C.1.12 * Utiliser une récurrence sur $k \leq n/2$ pour démontrer l'inégalité (C.6), et se servir de l'équation (C.3) pour l'étendre à tous les $k \leq n$.

C.1.13 * Utiliser l'approximation de Stirling pour démontrer que

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}} (1 + O(1/n)). \quad (\text{C.10})$$

C.1.14 * En dérivant la fonction d'entropie $H(\lambda)$, montrer qu'elle atteint un maximum en $\lambda = 1/2$. Que vaut $H(1/2)$?

C.1.15 * Montrer que, pour tout entier $n \geq 0$,

$$\sum_{k=0}^n \binom{n}{k} k = n2^{n-1}. \quad (\text{C.11})$$

C.2 PROBABILITÉS

Les probabilités sont un outil essentiel pour la conception et l'analyse des algorithmes probabilistes et randomisés. Cette section permet de réviser les bases de la théorie des probabilités.

On définit une probabilité en fonction d'un *espace d'épreuves* E , qui est un ensemble dont les éléments sont appelés *événements élémentaires*. Chaque événement élémentaire peut être vu comme un résultat possible d'une expérience. Pour l'expérience consistant à lancer deux pièces distinctes, on peut voir l'espace d'épreuves comme l'ensemble de toutes les chaînes d'ordre 2 (pile ou face) sur l'ensemble $\{F, P\}$:

$$E = \{FF, FP, PF, PP\}.$$

Un *événement* est un sous-ensemble ⁽¹⁾ de l'espace d'épreuves E . Par exemple, dans l'expérience consistant à lancer deux pièces, l'événement consistant à obtenir un face et un pile est $\{FP, PF\}$. L'événement E est appelé l'*événement certain*, et l'événement \emptyset est appelé l'*événement impossible*. On dit que deux événements A et B sont *mutuellement exclusifs* si $A \cap B = \emptyset$. On traite parfois un événement élémentaire $e \in E$ comme l'événement $\{e\}$. Par définition, tous les événements élémentaires sont mutuellement exclusifs.

a) Axiomes de probabilité

Une *distribution de probabilité* $\Pr\{\}$ sur un espace d'épreuves E est une correspondance entre les événements de E et les nombres réels telle que les *axiomes de probabilité* suivants soient satisfaits :

- 1) $\Pr\{A\} \geq 0$ pour un événement A quelconque.
- 2) $\Pr\{E\} = 1$.

(1) Pour une distribution de probabilité générale, il peut y avoir des sous-ensembles de l'espace d'épreuves E qui ne sont pas considérés comme des événements. Cette situation se produit généralement lorsque l'espace d'épreuves est infini non dénombrable. L'obligation principale est que l'ensemble des événements d'un espace d'épreuves soit fermé pour les opérations consistant à prendre le complément d'un événement, à former l'union d'une quantité finie ou dénombrable d'événements, et à prendre l'intersection d'une quantité finie ou dénombrable d'événements. La plupart des distributions de probabilité que nous verrons sont sur des espaces d'épreuves finis ou dénombrables, et l'on considérera en général que tous les sous-ensembles d'un espace d'épreuves sont des événements. Exception notable : la distribution de probabilité uniforme continue, qui sera présentée bientôt.

- 3) $\Pr \{A \cup B\} = \Pr \{A\} + \Pr \{B\}$ pour deux événements mutuellement exclusifs quelconques A et B . Plus généralement, pour toute séquence (finie ou infinie dénombrable) d'événements A_1, A_2, \dots mutuellement exclusifs deux à deux,

$$\Pr \left\{ \bigcup_i A_i \right\} = \sum_i \Pr \{A_i\} .$$

On appelle $\Pr \{A\}$ la **probabilité** de l'événement A . On remarque ici que l'axiome 2 correspond à une contrainte de normalisation : en réalité, rien n'impose de choisir la valeur 1 comme probabilité de l'événement certain, mis à part le fait qu'elle est à la fois naturelle et pratique.

Plusieurs résultats se déduisent immédiatement de ces axiomes et des fondations de la théorie des ensembles (voir section B.1). L'événement impossible \emptyset a la probabilité $\Pr \{\emptyset\} = 0$. Si $A \subseteq B$, alors $\Pr \{A\} \leq \Pr \{B\}$. En utilisant \bar{A} pour représenter l'événement $E - A$ (le **complément** de A), on a $\Pr \{\bar{A}\} = 1 - \Pr \{A\}$. Pour deux événements A et B quelconques,

$$\Pr \{A \cup B\} = \Pr \{A\} + \Pr \{B\} - \Pr \{A \cap B\} \quad (\text{C.12})$$

$$\leq \Pr \{A\} + \Pr \{B\} . \quad (\text{C.13})$$

En reprenant notre exemple du lancer de pièces, on suppose que chacun des quatre événements élémentaires a la probabilité $1/4$. La probabilité d'obtenir au moins un face est donc

$$\begin{aligned} \Pr \{FF, FP, PF\} &= \Pr \{FF\} + \Pr \{FP\} + \Pr \{PF\} \\ &= 3/4 . \end{aligned}$$

Autre façon de voir les choses : puisque la probabilité d'obtenir strictement moins qu'un face est $\Pr \{PP\} = 1/4$, la probabilité d'obtenir au moins une face est $1 - 1/4 = 3/4$.

b) Distributions discrètes

Une distribution de probabilité est **discrète** si elle est définie sur un espace d'épreuves fini ou infini dénombrable. Soit E l'espace d'épreuves. Alors, quel que soit l'événement A ,

$$\Pr \{A\} = \sum_{e \in A} \Pr \{e\} ,$$

puisque les événements élémentaires, en particuliers ceux de A sont mutuellement exclusifs. Si E est fini et que chaque événement élémentaire $e \in E$ a la probabilité

$$\Pr \{e\} = 1/|E| ,$$

on a la **distribution de probabilité uniforme** sur E . Dans ce type de cas, l'expérience est souvent décrite par « choisir un élément de E au hasard ».

Par exemple, considérons le lancer d'une *pièce non truquée*, pour laquelle la probabilité d'obtenir un face est la même que celle d'obtenir un pile, soit $1/2$. Si on lance la pièce n fois, on a la distribution de probabilité uniforme définie sur l'espace d'épreuves $E = \{F, P\}^n$, ensemble de taille 2^n . Chaque événement élémentaire de E peut être représenté par une chaîne de longueur n sur $\{F, P\}$, et chacun se produit avec la probabilité $1/2^n$. L'événement

$$A = \{\text{exactement } k \text{ faces et exactement } n - k \text{ piles}\}$$

est un sous-ensemble de E de taille $|A| = \binom{n}{k}$, puisqu'il existe $\binom{n}{k}$ chaînes de longueur n sur $\{F, P\}$ qui contiennent exactement k F. La probabilité de l'événement A est donc $\Pr \{A\} = \binom{n}{k} / 2^n$.

c) Distribution de probabilité uniforme continue

La distribution de probabilité uniforme continue est un exemple de distribution de probabilité dans laquelle les sous-ensembles de l'espace d'épreuves ne sont pas tous considérés comme des événements. La distribution de probabilité uniforme continue est définie sur un intervalle fermé $[a, b]$ de réels. Intuitivement, on souhaite que chaque point de l'intervalle $[a, b]$ soit « équiprobable ». Toutefois, c'est un ensemble de points non dénombrable, et si l'on donne à tous les points la même probabilité positive finie, on ne peut pas satisfaire simultanément les axiomes 2 et 3. Pour cette raison, on aimerait associer une probabilité uniquement à *certaines* des sous-ensembles de E de façon que les axiomes soient satisfaits pour ces événements.

Pour tout intervalle fermé $[c, d]$, où $a \leq c \leq d \leq b$, la **distribution de probabilité uniforme continue** définit la probabilité de l'événement $[c, d]$ comme

$$\Pr \{[c, d]\} = \frac{d - c}{b - a}.$$

Notez que, pour un point $x = [x, x]$ quelconque, la probabilité de x est 0. Si l'on supprime les extrémités d'un intervalle $[c, d]$, on obtient l'intervalle ouvert $]c, d[$. Comme $[c, d] = [c, c] \cup]c, d[$, l'axiome 3 nous donne $\Pr \{[c, d]\} = \Pr \{]c, d[\}$. En général, l'ensemble des événements pour la distribution de probabilité uniforme continue est un sous-ensemble quelconque de l'ensemble d'épreuves $[a, b]$ produit par une union finie ou infinie dénombrable d'intervalles fermés et ouverts.

d) Probabilité conditionnelle et indépendance

Il arrive qu'on ait une connaissance partielle *a priori* du résultat d'une expérience. Par exemple, supposons qu'un ami lance deux pièces de monnaie non truquées, et qu'il vous annonce qu'au moins une des pièces montre son côté face. Quelle est la probabilité pour que les deux pièces montrent leur côté face ? L'information donnée élimine la possibilité des deux piles. Les trois événements élémentaires restant sont équiprobables, on en déduit donc que chacun se produit avec une probabilité égale à $1/3$. Comme un seul de ces événements élémentaires montre deux faces, la réponse à notre question est : $1/3$.

La probabilité conditionnelle formalise la notion de connaissance partielle *a priori* du résultat d'une expérience. La **probabilité conditionnelle** d'un événement A , en supposant qu'un autre événement B se produit, se définit par

$$\Pr \{A | B\} = \frac{\Pr \{A \cap B\}}{\Pr \{B\}} \quad (\text{C.14})$$

pourvu que $\Pr \{B\} \neq 0$. (« $\Pr \{A | B\}$ » se lit « probabilité de A sachant B ».) Intuitivement, puisqu'on sait que l'événement B se produit, l'événement traduisant que A se produit aussi est $A \cap B$. Autrement dit, $A \cap B$ est l'ensemble des résultats pour lesquels à la fois A et B se produisent. Puisque le résultat est l'un des événements élémentaires de B , on normalise les probabilités de tous les événements élémentaires de B en les divisant par $\Pr \{B\}$, de manière que leur somme soit égale à 1. La probabilité conditionnelle de A sachant B est, par suite, le rapport de la probabilité de l'événement $A \cap B$ sur la probabilité de l'événement B . Dans l'exemple ci-dessus, A est l'événement traduisant que les deux pièces montrent leur côté face, et B est l'événement traduisant qu'au moins une pièce est un face. Dans ce cas, $\Pr \{A | B\} = (1/4)/(3/4) = 1/3$.

Deux événements sont **indépendants** si

$$\Pr \{A \cap B\} = \Pr \{A\} \Pr \{B\} , \quad (\text{C.15})$$

ce qui équivaut, si $\Pr \{B\} \neq 0$, à la condition

$$\Pr \{A | B\} = \Pr \{A\} .$$

Par exemple, on suppose que deux pièces non truquées sont lancées et que les résultats sont indépendants. Dans ce cas, la probabilité d'obtenir deux faces est $(1/2)(1/2) = 1/4$. Supposons maintenant qu'un événement est que la première pièce montre son côté face, et que l'autre événement est que les pièces retombent différemment. Chacun de ces événements se produit avec la probabilité $1/2$, et la probabilité pour que les deux événements se produisent est $1/4$; donc, d'après la définition de l'indépendance, les événements sont indépendants (bien que l'on puisse penser que les deux événements dépendent de la première pièce). Enfin, supposons que les pièces soient soudées l'une à l'autre, de façon qu'elles retombent toutes les deux sur face ou toutes les deux sur pile, et que les deux possibilités soient équiprobables. Alors, la probabilité pour que chaque pièce montre son côté face est $1/2$, mais la probabilité pour qu'elles montrent toutes les deux leur côté face est $1/2 \neq (1/2)(1/2)$. Dans ce cas, l'événement traduisant qu'une pièce tombe sur face, et celui traduisant que l'autre tombe sur face ne sont pas indépendants.

On dit d'un ensemble A_1, A_2, \dots, A_n d'événements qu'ils sont **indépendants deux à deux** si

$$\Pr \{A_i \cap A_j\} = \Pr \{A_i\} \Pr \{A_j\}$$

pour tout $1 \leq i < j \leq n$. On dit que ces événements sont (**mutuellement**) **indépendants** si chaque k -sous-ensemble $A_{i_1}, A_{i_2}, \dots, A_{i_k}$ de la collection, où $2 \leq k \leq n$ et $1 \leq i_1 < i_2 < \dots < i_k \leq n$, satisfait à

$$\Pr \{A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_k}\} = \Pr \{A_{i_1}\} \Pr \{A_{i_2}\} \dots \Pr \{A_{i_k}\} .$$

Par exemple, supposons qu'on lance deux pièces non truquées. Soit A_1 l'événement traduisant le fait que la première pièce tombe sur face ; soit A_2 l'événement traduisant le fait que la seconde pièce tombe sur face ; et soit A_3 l'événement traduisant que les deux pièces retombent différemment. On a

$$\begin{aligned}\Pr \{A_1\} &= 1/2, \\ \Pr \{A_2\} &= 1/2, \\ \Pr \{A_3\} &= 1/2, \\ \Pr \{A_1 \cap A_2\} &= 1/4, \\ \Pr \{A_1 \cap A_3\} &= 1/4, \\ \Pr \{A_2 \cap A_3\} &= 1/4, \\ \Pr \{A_1 \cap A_2 \cap A_3\} &= 0.\end{aligned}$$

Puisque, pour $1 \leq i < j \leq 3$, on a $\Pr \{A_i \cap A_j\} = \Pr \{A_i\} \Pr \{A_j\} = 1/4$, les événements A_1 , A_2 , et A_3 sont indépendants deux à deux. Mais, comme $\Pr \{A_1 \cap A_2 \cap A_3\} = 0$ et $\Pr \{A_1\} \Pr \{A_2\} \Pr \{A_3\} = 1/8 \neq 0$, les événements ne sont pas mutuellement indépendants.

e) Théorème de Bayes

De la définition de la probabilité conditionnelle (C.14) et de la loi commutative $A \cap B = B \cap A$, il s'ensuit que, pour deux événements A et B ayant chacun une probabilité non nulle, l'on a

$$\begin{aligned}\Pr \{A \cap B\} &= \Pr \{B\} \Pr \{A | B\} \\ &= \Pr \{A\} \Pr \{B | A\}.\end{aligned}\tag{C.16}$$

En résolvant pour $\Pr \{A | B\}$, on obtient

$$\Pr \{A | B\} = \frac{\Pr \{A\} \Pr \{B | A\}}{\Pr \{B\}},\tag{C.17}$$

résultat connu en tant que **théorème de Bayes**. Le dénominateur $\Pr \{B\}$ est une constante normalisatrice que l'on peut reformuler de la manière suivante. Puisque $B = (B \cap A) \cup (B \cap \bar{A})$ et $B \cap A$ et $B \cap \bar{A}$ sont des événements mutuellement exclusifs,

$$\begin{aligned}\Pr \{B\} &= \Pr \{B \cap A\} + \Pr \{B \cap \bar{A}\} \\ &= \Pr \{A\} \Pr \{B | A\} + \Pr \{\bar{A}\} \Pr \{B | \bar{A}\}.\end{aligned}$$

En substituant dans l'équation (C.17), on obtient une forme équivalente du théorème de Bayes :

$$\Pr \{A | B\} = \frac{\Pr \{A\} \Pr \{B | A\}}{\Pr \{A\} \Pr \{B | A\} + \Pr \{\bar{A}\} \Pr \{B | \bar{A}\}}.$$

Le théorème de Bayes peut simplifier le calcul des probabilités conditionnelles. Par exemple, supposons qu'on dispose d'une pièce non truquée et d'une pièce truquée qui retombe toujours sur face. On réalise l'expérience consistant en trois événements

indépendants : l'une des deux pièces est choisie au hasard, la pièce est lancée une fois, puis lancée une nouvelle fois. On suppose que la pièce choisie est retombée deux fois sur face. Quelle est la probabilité pour qu'elle soit truquée ?

On résout ce problème à l'aide du théorème de Bayes. Soit A l'événement traduisant que la pièce truquée est choisie, et soit B l'événement traduisant que la pièce retombe deux fois sur face. On souhaite déterminer $\Pr\{A \mid B\}$. On a $\Pr\{A\} = 1/2$, $\Pr\{B \mid A\} = 1$, $\Pr\{\bar{A}\} = 1/2$ et $\Pr\{B \mid \bar{A}\} = 1/4$; d'où

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} \\ &= 4/5.\end{aligned}$$

Exercices

C.2.1 Démontrer *l'inégalité de Boole* : Pour toute séquence finie ou infinie dénombrable d'événements A_1, A_2, \dots ,

$$\Pr\{A_1 \cup A_2 \cup \dots\} \leq \Pr\{A_1\} + \Pr\{A_2\} + \dots \quad (\text{C.18})$$

C.2.2 Le Professeur Rosencrantz lance une fois une pièce non truquée. Le Professeur Guildenstern lance deux fois une pièce non truquée. Quelle est la probabilité pour que le Professeur Rosencrantz obtienne plus de faces que le Professeur Guildenstern ?

C.2.3 On bat un paquet de 10 cartes, portant chacune un numéro distinct de 1 à 10. Trois cartes sont retirées du paquet l'une après l'autre. Quelle est la probabilité pour que les trois cartes apparaissent dans un ordre croissant ?

C.2.4 * Décrire une procédure qui prend en entrée deux entiers a et b tels que $0 < a < b$ et qui, en se basant sur des lancers de pièce non truquée, produit en sortie des faces avec une probabilité a/b et des piles avec une probabilité $(b-a)/b$. Donner une borne pour le nombre attendu de lancers de pièces, qui doit être $O(1)$. (*Conseil* : Représenter a/b en binaire.)

C.2.5 Démontrer que

$$\Pr\{A \mid B\} + \Pr\{\bar{A} \mid B\} = 1.$$

C.2.6 Démontrer que, pour tout ensemble d'événements A_1, A_2, \dots, A_n ,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \dots \cap A_n\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_n \mid A_1 \cap A_2 \cap \dots \cap A_{n-1}\}.\end{aligned}$$

C.2.7 * Montrer comment construire un ensemble de n événements indépendants deux à deux, tel qu'aucun sous-ensemble de $k > 2$ éléments ne soit mutuellement indépendant.

C.2.8 * Deux événements A et B sont *conditionnellement indépendants*, sachant C , si

$$\Pr \{A \cap B \mid C\} = \Pr \{A \mid C\} \cdot \Pr \{B \mid C\} .$$

Donner un exemple simple, mais non trivial, de deux événements qui ne sont pas indépendants mais qui sont conditionnellement indépendants, sachant un troisième événement.

C.2.9 * Vous êtes candidat à un jeu télévisé où le cadeau est caché derrière l'un des trois rideaux qui sont en face de vous. Vous gagnerez le cadeau si vous choisissez le bon rideau. Après que vous avez choisi un rideau, mais avant que celui-ci ne soit relevé, l'animateur lève l'un des autres rideaux, en sachant qu'il va dévoiler une cache vide. Il vous demande alors si vous souhaitez abandonner votre choix initial pour le dernier rideau. Dans quelle mesure vos chances seront-elles modifiées si vous changez de choix ?

C.2.10 * Un directeur de prison a choisi au hasard un prisonnier parmi trois pour le libérer. Les deux autres seront exécutés. Le gardien sait lequel sortira, mais a l'interdiction de donner à un prisonnier des informations sur son statut. Appelons les prisonniers X , Y et Z . Le prisonnier X demande au garde, en privé, lequel de Y ou de Z sera exécuté, arguant que, puisqu'il sait déjà que celui qui mourra est l'un d'eux, le gardien en lui répondant ne révélera aucune information sur son propre statut. Le gardien annonce à X que Y sera exécuté. Le prisonnier X est soulagé, puisqu'il imagine à présent que soit lui, soit le prisonnier Z sera libéré, ce qui signifie que sa probabilité de sortie est passée maintenant à $1/2$. A-t-il raison, ou ses chances sont-elles encore de $1/3$? Dire pourquoi.

C.3 VARIABLES ALÉATOIRES DISCRÈTES

Une *variable aléatoire (discrète)* X est une fonction d'un espace d'épreuves fini ou infini dénombrable E vers les nombres réels. Elle associe un nombre réel à chaque résultat possible d'une expérience, ce qui nous permet de travailler avec la distribution de probabilité engendrée sur l'ensemble de nombres résultant. Les variables aléatoires peuvent aussi être définies pour des espaces d'épreuves infinis non dénombrables, mais elles posent des problèmes techniques dont il est inutile de s'encombrer pour notre propos. Nous supposons donc que les variables aléatoires sont discrètes.

Pour une variable aléatoire X et un nombre réel x , on définit l'événement $X = x$ par $\{e \in E : X(e) = x\}$; donc,

$$\Pr \{X = x\} = \sum_{\{e \in E : X(e) = x\}} \Pr \{e\} .$$

La fonction

$$f(x) = \Pr \{X = x\}$$

est la *fonction de densité de probabilité* de la variable aléatoire X . D'après les axiomes de probabilité, $\Pr \{X = x\} \geq 0$ et $\sum_x \Pr \{X = x\} = 1$.

Comme exemple, considérons l'expérience de faire rouler une paire de dés ordinaires à six faces. Il existe 36 événements élémentaires possibles dans l'espace

d'épreuves. On suppose que la distribution de probabilité est uniforme, de sorte que chaque événement élémentaire $e \in E$ est équiprobable : $\Pr\{e\} = 1/36$. On définit la variable aléatoire X comme étant le *maximum* des deux valeurs montrées par les dés. On a $\Pr\{X = 3\} = 5/36$, puisque X assigne une valeur de 3 à 5 événements élémentaires parmi les 36 possibles, c'est-à-dire (1, 3), (2, 3), (3, 3), (3, 2), et (3, 1).

On définit souvent plusieurs variables aléatoires sur le même espace d'épreuves. Si X et Y sont des variables aléatoires, la fonction

$$f(x, y) = \Pr\{X = x \text{ et } Y = y\}$$

est la *fonction de densité de probabilité conjointe* de X et Y . Pour une valeur fixée y ,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ et } Y = y\},$$

et de la même manière, pour une valeur fixée x ,

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ et } Y = y\}.$$

En utilisant la définition (C.14) de la probabilité conditionnelle, on a

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ et } Y = y\}}{\Pr\{Y = y\}}.$$

On dit que deux variables aléatoires X et Y sont *indépendantes* si, pour tout x et y , les événements $X = x$ et $Y = y$ sont indépendants ou, de manière équivalente, si pour tout x et y , on a $\Pr\{X = x \text{ et } Y = y\} = \Pr\{X = x\} \Pr\{Y = y\}$.

Étant donné un ensemble de variables aléatoires définies sur le même espace d'épreuves, on peut définir de nouvelles variables aléatoires comme sommes, produits ou autres fonctions des variables initiales.

a) *Espérance d'une variable aléatoire*

Le résumé le plus simple et le plus utile qu'on puisse faire de la distribution d'une variable aléatoire est la « moyenne » des valeurs qu'elle prend. L'*espérance* (ou encore, la *valeur attendue*, ou *moyenne*) d'une variable aléatoire discrète X est

$$E[X] = \sum_x x \Pr\{X = x\}, \quad (\text{C.19})$$

qui est bien définie si la somme est finie ou converge absolument. Parfois, l'espérance de X est notée μ_X ou, lorsque la variable aléatoire s'impose par le contexte, simplement μ .

Considérons un jeu dans lequel on lance deux pièces non truquées. On gagne 3 euros pour chaque face, mais on perd 2 euros pour chaque pile. L'espérance de la variable aléatoire X représentant les gains est

$$\begin{aligned} E[X] &= 6 \cdot \Pr\{2 F\} + 1 \cdot \Pr\{1 F, 1 P\} - 4 \cdot \Pr\{2 P\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1. \end{aligned}$$

L'espérance de la somme de deux variables aléatoires est la somme de leurs espérances, c'est-à-dire

$$E[X + Y] = E[X] + E[Y], \quad (\text{C.20})$$

pourvu que $E[X]$ et $E[Y]$ soient définies. Cette propriété, qui est dite **linéarité de l'espérance** et qui reste vraie même si X et Y ne sont pas indépendants, s'étend aux sommations finies et aux séries absolument convergentes d'espérances. La linéarité est la propriété fondamentale pour faire des analyses probabilistes à l'aide de variables indicatrices (voir section 5.2).

Si X est une variable aléatoire, toute fonction $g(x)$ définit une nouvelle variable aléatoire $g(X)$. Si l'espérance de $g(X)$ est définie, alors

$$E[g(X)] = \sum_x g(x) \Pr\{X = x\}.$$

En posant $g(x) = ax$, on a pour une constante a quelconque,

$$E[aX] = aE[X]. \quad (\text{C.21})$$

Par suite, les espérances sont linéaires : pour deux variables aléatoires quelconques X et Y , et une constante quelconque a ,

$$E[aX + Y] = aE[X] + E[Y]. \quad (\text{C.22})$$

Lorsque deux variables aléatoires X et Y sont indépendantes et que chacune a une espérance définie,

$$\begin{aligned} E[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ et } Y = y\} \\ &= \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} \\ &= \left(\sum_x x \Pr\{X = x\} \right) \left(\sum_y y \Pr\{Y = y\} \right) \\ &= E[X]E[Y]. \end{aligned}$$

En général, quand n variables aléatoires X_1, X_2, \dots, X_n sont mutuellement indépendantes,

$$E[X_1 X_2 \cdots X_n] = E[X_1] E[X_2] \cdots E[X_n]. \quad (\text{C.23})$$

Quand une variable aléatoire X prend des valeurs dans l'ensemble des entiers naturels $\mathbf{N} = \{0, 1, 2, \dots\}$, il existe une jolie formule pour son espérance :

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} , \end{aligned} \tag{C.24}$$

puisque chaque terme $\Pr\{X \geq i\}$ est ajouté i fois et soustrait $i - 1$ fois (sauf $\Pr\{X \geq 0\}$ qui est ajouté 0 fois et jamais soustrait).

Quand on applique une fonction convexe $f(x)$ à une variable aléatoire X , l'**inégalité de Jensen** nous donne

$$E[f(X)] \geq f(E[X]) , \tag{C.25}$$

à condition que les espérances existent et soient finies. (Une fonction $f(x)$ est **convexe** si, pour tout x et y et pour tout $0 \leq \lambda \leq 1$, on a $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$.)

b) Variance et écart-type

L'espérance d'une variable aléatoire ne nous dit rien sur la « dispersion » des valeurs de la variable. Par exemple, si l'on a des variables aléatoires X et Y pour lesquelles $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ et $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$, alors $E[X]$ et $E[Y]$ valent tous les deux $1/2$, alors même que les valeurs réelles prises par Y sont plus éloignées de la moyenne que les valeurs réelles prises par X .

La notion de variance exprime mathématiquement les écarts probables entre les valeurs d'une variable aléatoire et la valeur moyenne. La **variance** d'une variable aléatoire X ayant pour moyenne $E[X]$ est

$$\begin{aligned} \text{Var}[X] &= E[(X - E[X])^2] \\ &= E[X^2 - 2XE[X] + E^2[X]] \\ &= E[X^2] - 2E[XE[X]] + E^2[X] \\ &= E[X^2] - 2E^2[X] + E^2[X] \\ &= E[X^2] - E^2[X] . \end{aligned} \tag{C.26}$$

On justifie les égalités $E[E^2[X]] = E^2[X]$ et $E[XE[X]] = E^2[X]$ par le fait que $E[X]$ n'est pas une variable aléatoire, mais tout simplement un nombre réel, ce qui signifie que l'équation (C.21) s'applique (avec $a = E[X]$). L'équation (C.26) peut être réécrite pour obtenir une expression de l'espérance du carré d'une variable aléatoire :

$$E[X^2] = \text{Var}[X] + E^2[X] . \tag{C.27}$$

La variance d'une variable aléatoire X et la variance de aX sont reliées (voir exercice C.3.10) :

$$\text{Var}[aX] = a^2 \text{Var}[X] .$$

Lorsque X et Y sont des variables aléatoires indépendantes,

$$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y] .$$

En général, si n variables aléatoires X_1, X_2, \dots, X_n sont indépendantes deux à deux, on a :

$$\text{Var}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \text{Var}[X_i] . \quad (\text{C.28})$$

L'**écart-type** d'une variable aléatoire X est la racine carrée positive de la variance de X . L'écart-type d'une variable aléatoire X est parfois noté σ_X ou simplement σ lorsque la variable aléatoire X est donnée par le contexte. Avec cette notation, la variance de X est notée σ^2 .

Exercices

C.3.1 Deux dés ordinaires à 6 faces sont lancés. Quelle est l'espérance de la somme des deux valeurs apparues ? Quelle est l'espérance du maximum des deux valeurs apparues ?

C.3.2 Un tableau $A[1..n]$ contient n nombres distincts qui sont ordonnés de façon aléatoire, toutes les permutations possibles des n nombres étant équiprobables. Quelle est l'espérance de l'indice de l'élément maximal du tableau ? Quelle est l'espérance de l'indice de l'élément minimal ?

C.3.3 Un jeu de kermesse utilise trois dés enfermés dans une cage. Un joueur peut parier un Euro sur un numéro quelconque entre 1 et 6. La cage est secouée, et le gain est réparti de la façon suivante : si le numéro du joueur n'apparaît sur aucun des dés, il perd sa mise. Sinon, si son numéro apparaît sur exactement k dés, pour $k = 1, 2, 3$, il récupère sa mise et gagne k Euros supplémentaires. Combien peut-il s'attendre à gagner s'il joue une seule fois ?

C.3.4 Prouver que, si X et Y sont des variables aléatoires non négatives, alors

$$E[\max(X, Y)] \leq E[X] + E[Y] .$$

C.3.5 ★ Soient X et Y des variables aléatoires indépendantes. Démontrer que $f(X)$ et $g(Y)$ sont indépendantes quelles que soient les fonctions f et g .

C.3.6 ★ Soit X une variable aléatoire non négative ; on suppose que $E[X]$ est définie. Prouver l'**inégalité de Markov** :

$$\Pr\{X \geq t\} \leq E[X]/t \quad (\text{C.29})$$

pour tout $t > 0$.

C.3.7 ★ Soit E un espace d'épreuves, et soient X et X' des variables aléatoires telles que $X(e) \geq X'(e)$ pour tout $e \in E$. Démontrer que, pour toute constante réelle t , on a

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\} .$$

C.3.8 Des deux valeurs suivantes, quelle est la plus grande : l'espérance du carré d'une variable aléatoire, ou le carré de son espérance ?

C.3.9 Montrer que, pour toute variable aléatoire X qui prend seulement les valeurs 0 et 1, on a $\text{Var}[X] = E[X]E[1 - X]$.

C.3.10 Démontrer que $\text{Var}[aX] = a^2 \text{Var}[X]$ d'après la définition (C.26) de la variance.

C.4 DISTRIBUTIONS GÉOMÉTRIQUE ET BINOMIALE

Un lancer de pièce est une instance d'une *épreuve de Bernoulli*, qui se définit comme une expérience ayant seulement deux résultats possibles : le *succès*, qui se produit avec la probabilité p , et l'*échec*, qui se produit avec une probabilité $q = 1 - p$. Quand on parle d'*épreuves de Bernoulli* collectivement, on veut signifier que les épreuves sont mutuellement indépendantes et que, sauf contre-ordre explicite, chacune a la même probabilité de succès p . Deux distributions importantes sont induites par les épreuves de Bernoulli : la distribution géométrique et la distribution binomiale.

a) Distribution géométrique

On suppose qu'on a une séquence d'épreuves de Bernoulli, chacune avec une probabilité de succès égale à p et une probabilité d'échec égale à $q = 1 - p$. Combien d'essais faut-il faire avant d'obtenir un succès ? Attribuons à la variable aléatoire X le nombre d'essais nécessaires pour obtenir un succès. X prend alors ses valeurs dans l'ensemble $\{1, 2, \dots\}$ et, pour $k \geq 1$,

$$\Pr\{X = k\} = q^{k-1}p , \tag{C.30}$$

puisque l'on a $k - 1$ échecs avant le premier succès. Une distribution de probabilité qui satisfait à l'équation (C.30) est dite *distribution géométrique*. La figure C.1 illustre ce type de distribution.

En supposant que $q < 1$, l'espérance d'une distribution géométrique peut se calculer à l'aide de l'identité (A.8) :

$$E[X] = \sum_{k=1}^{\infty} kq^{k-1}p = \frac{p}{q} \sum_{k=0}^{\infty} kq^k = \frac{p}{q} \cdot \frac{q}{(1-q)^2} = 1/p . \tag{C.31}$$

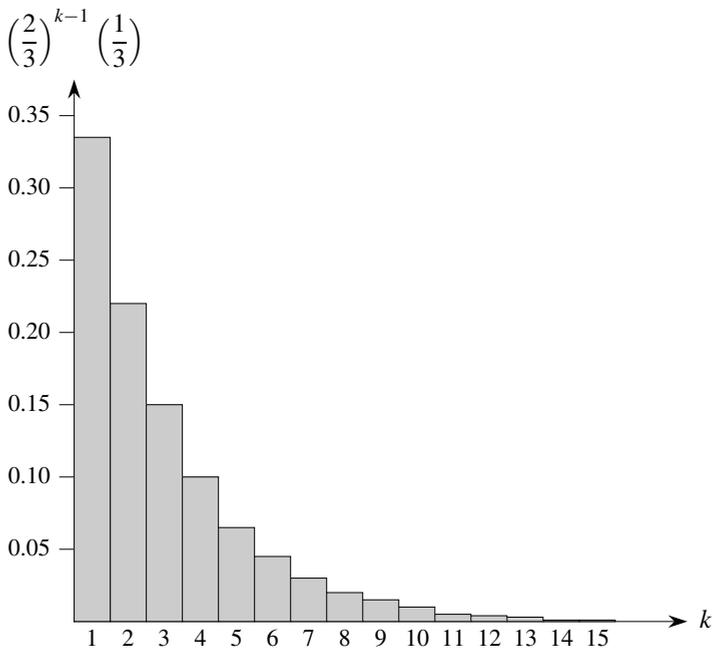


Figure C.1 Une distribution géométrique avec une probabilité de succès $p = 1/3$ et une probabilité $q = 1 - p$ d'échec. L'espérance de la distribution est $1/p = 3$.

Donc, en moyenne, il faut $1/p$ tentatives avant d'obtenir un succès, ce qu'on pouvait pressentir intuitivement. La variance, qui peut-être calculée de la même façon, vaut d'après l'exercice A.1.3

$$\text{Var}[X] = q/p^2. \quad (\text{C.32})$$

Par exemple, on suppose qu'on lance à plusieurs reprises deux dés, jusqu'à obtenir soit sept, soit onze. Parmi les 36 résultats possibles, 6 produisent la valeur sept et 2 produisent onze. Donc, la probabilité de succès est $p = 8/36 = 2/9$, et il faut lancer $1/p = 9/2 = 4,5$ fois en moyenne pour obtenir un sept ou un onze.

b) Distribution binomiale

Combien de succès obtient-on durant n épreuves de Bernoulli, où un succès survient avec la probabilité p et un échec avec la probabilité $q = 1 - p$? On définit la variable aléatoire X comme représentant le nombre de succès au cours de n tentatives. X prend alors ses valeurs dans l'intervalle $\{0, 1, \dots, n\}$, et pour $k = 0, \dots, n$,

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \quad (\text{C.33})$$

puisque'il existe $\binom{n}{k}$ manière de choisir k succès parmi n tentatives, et la probabilité d'apparition de chacun est $p^k q^{n-k}$. Une distribution de probabilité qui satisfait à

l'équation (C.33) est appelée **distribution binomiale**. Pour des raisons pratiques, on définit la famille des distributions binomiales avec la notation

$$b(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}. \quad (\text{C.34})$$

La figure C.2 illustre une distribution binomiale. L'adjectif « binomiale » vient du fait que (C.34) est le k ième terme du développement de $(p + q)^n$. En conséquence, puisque $p + q = 1$,

$$\sum_{k=0}^n b(k; n, p) = 1, \quad (\text{C.35})$$

comme le veut l'axiome 2 des axiomes de probabilité.

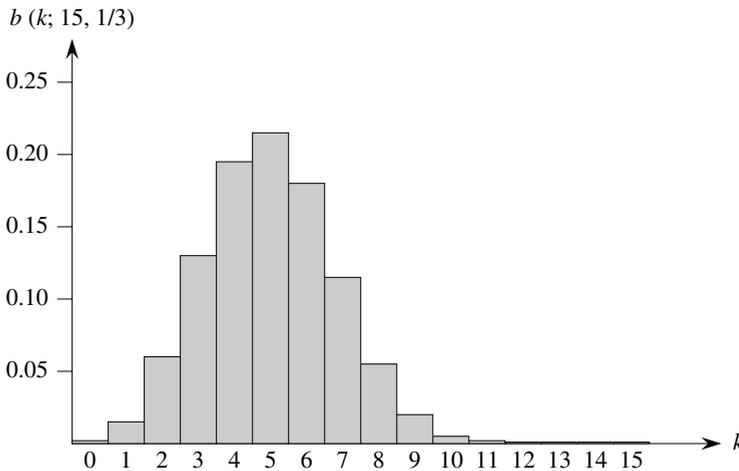


Figure C.2 Distribution binomiale $b(k; 15, 1/3)$ résultant de $n = 15$ épreuves de Bernoulli, chacune ayant une probabilité $p = 1/3$ de succès. L'espérance de la distribution est $np = 5$.

On peut calculer l'espérance d'une variable aléatoire ayant une distribution binomiale à partir des équations (C.8) et (C.35). Soit X une variable aléatoire obéissant à la distribution binomiale $b(k; n, p)$, et soit $q = 1 - p$. D'après la définition de l'espérance, on a

$$\begin{aligned} E[X] &= \sum_{k=0}^n k \Pr\{X = k\} = \sum_{k=0}^n k b(k; n, p) = \sum_{k=1}^n k \binom{n}{k} p^k q^{n-k} \\ &= np \sum_{k=1}^n \binom{n-1}{k-1} p^{k-1} q^{n-k} = np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\ &= np \sum_{k=0}^{n-1} b(k; n-1, p) = np. \end{aligned} \quad (\text{C.36})$$

En se servant de la linéarité de l'espérance, on peut obtenir le même résultat en faisant nettement moins appel à l'algèbre. Soit X_i la variable aléatoire décrivant le nombre de succès au cours de la i ème tentative. Alors $E[X_i] = p \cdot 1 + q \cdot 0 = p$ et, d'après la linéarité de l'espérance (C.20), le nombre de succès attendu pour n tentatives est

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n p = np. \quad (\text{C.37})$$

On peut utiliser la même approche pour calculer la variance de la distribution. D'après l'équation (C.26), on a $\text{Var}[X_i] = E[X_i^2] - E^2[X_i]$. Puisque X_i ne peut prendre que les valeurs 0 et 1, on a $E[X_i^2] = E[X_i] = p$; d'où

$$\text{Var}[X_i] = p - p^2 = pq. \quad (\text{C.38})$$

Pour calculer la variance de X , on utilise l'indépendance des n épreuves; ainsi, d'après l'équation (C.28),

$$\text{Var}[X] = \text{Var}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \text{Var}[X_i] = \sum_{i=1}^n pq = npq. \quad (\text{C.39})$$

Comme on peut le voir dans la figure C.2, la distribution binomiale $b(k; n, p)$ croît, quand k augmente de 0 à n , jusqu'à atteindre la valeur moyenne np , puis elle décroît. On peut prouver que la distribution se comporte toujours de cette manière en s'intéressant au rapport des termes consécutifs :

$$\begin{aligned} \frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} = \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} = \frac{(n-k+1)p}{kq} \\ &= 1 + \frac{(n+1)p - k}{kq}. \end{aligned} \quad (\text{C.40})$$

Ce rapport est plus grand que 1 précisément lorsque $(n+1)p - k$ est positif. En conséquence, $b(k; n, p) > b(k-1; n, p)$ pour $k < (n+1)p$ (la distribution croît) et $b(k; n, p) < b(k-1; n, p)$ pour $k > (n+1)p$ (la distribution décroît). Si $k = (n+1)p$ est un entier, alors $b(k; n, p) = b(k-1; n, p)$, et la distribution présente donc deux maxima : en $k = (n+1)p$ et en $k-1 = (n+1)p - 1 = np - q$. Sinon, elle atteint un maximum pour l'unique entier k se trouvant dans l'intervalle $np - q < k < (n+1)p$.

Le lemme suivant donne une majorant pour la distribution binomiale.

Lemme C.1 Soit $n \geq 0$, soit $0 < p < 1$, soit $q = 1 - p$ et soit $0 \leq k \leq n$. On a

$$b(k; n, p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

Démonstration : D'après l'équation (C.6), on a

$$\begin{aligned} b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \\ &\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}. \end{aligned}$$

□

Exercices

C.4.1 Vérifier l'axiome 2 des axiomes des probabilités pour la distribution géométrique.

C.4.2 Combien de fois en moyenne doit-on lancer 6 pièces non truquées avant d'obtenir 3 faces et 3 piles ?

C.4.3 Montrer que $b(k; n, p) = b(n - k; n, q)$, où $q = 1 - p$.

C.4.4 Montrer que la valeur du maximum de la distribution binomiale $b(k; n, p)$ est approximativement $1/\sqrt{2\pi npq}$, où $q = 1 - p$.

C.4.5 ★ Montrer que la probabilité de n'avoir aucun succès au cours de n épreuves de Bernoulli, chacune ayant $p = 1/n$, est environ $1/e$. Montrer que la probabilité d'avoir exactement un succès est également de $1/e$ environ.

C.4.6 ★ Le Professeur Rosencrantz lance une pièce non truquée n fois, et le professeur Guildenstern fait de même. Montrer que la probabilité qu'ils obtiennent le même nombre de faces est $\binom{2n}{n}/4^n$. (*conseil* : Pour le professeur Rosencrantz, un succès sera un face ; pour le Professeur Guildenstern, un succès sera un pile.) Utiliser votre démonstration pour vérifier l'identité

$$\sum_{k=0}^n \binom{n}{k^2} = \binom{2n}{n}.$$

C.4.7 ★ Montrer que, pour $0 \leq k \leq n$,

$$b(k; n, 1/2) \leq 2^{nH(k/n)-n},$$

où $H(x)$ est la fonction d'entropie (C.7).

C.4.8 ★ On considère n épreuves de Bernoulli où, pour $i = 1, 2, \dots, n$, la i ème épreuve a une probabilité de succès p_i ; soit X la variable aléatoire représentant le nombre de succès total. Soit $p \geq p_i$ pour tout $i = 1, 2, \dots, n$. Démontrer que pour $1 \leq k \leq n$,

$$\Pr \{X < k\} \leq \sum_{i=0}^{k-1} b(i; n, p).$$

C.4.9 * Soit X la variable aléatoire pour le nombre total de succès dans un ensemble A de n épreuves de Bernoulli où la i ème épreuve a une probabilité de succès p_i , et soit X' la variable aléatoire représentant le nombre de succès total dans un second ensemble A' de n épreuves de Bernoulli où la i ème épreuve a une probabilité de succès $p'_i \geq p_i$. Démontrer que, pour $0 \leq k \leq n$,

$$\Pr \{X' \geq k\} \geq \Pr \{X \geq k\} .$$

(conseil : Montrer comment obtenir les épreuves de Bernoulli de A' par une expérience impliquant les épreuves de A , puis utiliser le résultat de l'exercice C.3.7.)

C.5^{*} QUEUES DE LA DISTRIBUTION BINOMIALE

Il est souvent plus intéressant de connaître la probabilité d'avoir au moins, ou au plus, k succès au cours de n épreuves de Bernoulli ayant chacune une probabilité de succès p , que de connaître la probabilité d'avoir exactement k succès. Dans cette section, on s'intéresse aux **queues** de la distribution binomiale : les deux régions de la distribution $b(k; n, p)$ qui sont éloignées de la moyenne np . On prouvera plusieurs bornes importantes pour (la somme de tous les termes d') une queue.

On commencera par fournir une borne sur la queue de droite de la distribution $b(k; n, p)$. Les bornes pour la queue de gauche peuvent être déterminées en inversant les rôles des succès et des échecs.

Théorème C.2 *Considérons une séquence de n épreuves de Bernoulli pour lesquelles le succès se produit avec une probabilité p . Soit X la variable aléatoire représentant le nombre de total de succès. Dans ce cas, pour $0 \leq k \leq n$, la probabilité d'obtenir au moins k succès est*

$$\begin{aligned} \Pr \{X \geq k\} &= \sum_{i=k}^n b(i; n, p) \\ &\leq \binom{n}{k} p^k . \end{aligned}$$

Démonstration : Pour $S \subseteq \{1, 2, \dots, n\}$, soit A_S l'événement « la i ème épreuve est un succès pour tout $i \in S$ ». On a manifestement $\Pr \{A_S\} = p^k$ si $|S| = k$. On a

$$\begin{aligned} \Pr \{X \geq k\} &= \Pr \{ \text{il existe } S \subseteq \{1, 2, \dots, n\} : |S| = k \text{ et } A_S \} \\ &= \Pr \left\{ \bigcup_{S \subseteq \{1, 2, \dots, n\}; |S|=k} A_S \right\} \\ &\leq \sum_{S \subseteq \{1, 2, \dots, n\}; |S|=k} \Pr \{A_S\} \\ &= \binom{n}{k} p^k , \end{aligned}$$

L'inégalité découle de l'inégalité de Boole (C.18). □

Le corollaire suivant énonce le même théorème pour la queue de gauche. En général, on laissera au lecteur le soin d'adapter les bornes d'une queue à l'autre.

Corollaire C.3 *On considère une séquence de n épreuves de Bernoulli, où le succès est obtenu avec une probabilité p . Si X est la variable aléatoire désignant le nombre total de succès, alors pour $0 \leq k \leq n$, la probabilité d'avoir au plus k succès est*

$$\Pr \{X \leq k\} = \sum_{i=0}^k b(i; n, p) \leq \binom{n}{n-k} (1-p)^{n-k} = \binom{n}{k} (1-p)^{n-k}.$$

Notre prochaine borne concerne la queue de gauche de la distribution binomiale. Son corollaire montre que, loin de la moyenne, la queue de gauche diminue de manière exponentielle.

Théorème C.4 *On considère une séquence de n épreuves de Bernoulli, où le succès est obtenu avec une probabilité p et l'échec avec une probabilité $q = 1 - p$. Soit X la variable aléatoire représentant le nombre total de succès. Alors, pour $0 < k < np$, la probabilité d'obtenir moins de k succès est :*

$$\Pr \{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < \frac{kq}{np - k} b(k; n, p).$$

Démonstration : On borne la série $\sum_{i=0}^{k-1} b(i; n, p)$ par une série géométrique en employant la technique de la section A.2, page 1027. Pour $i = 1, 2, \dots, k$, on a d'après l'équation C.40,

$$\frac{b(i-1; n, p)}{b(i; n, p)} = \frac{iq}{(n-i+1)p} < \frac{iq}{(n-i)p} \leq \frac{kq}{(n-k)p}.$$

Si l'on fait

$$\begin{aligned} x &= \frac{kq}{(n-k)p} \\ &< \frac{kq}{(n-np)p} \\ &= \frac{kq}{nqp} \\ &= \frac{k}{np} \\ &< 1, \end{aligned}$$

il s'ensuit que

$$b(i-1; n, p) < x b(i; n, p)$$

pour $0 < i \leq k$. En appliquant successivement cette inégalité $k-i$ fois, on obtient

$$b(i; n, p) < x^{k-i} b(k; n, p)$$

pour $0 \leq i < k$; d'où

$$\begin{aligned} \sum_{i=0}^{k-1} b(i; n, p) &< \sum_{i=0}^{k-1} x^{k-i} b(k; n, p) \\ &< b(k; n, p) \sum_{i=1}^{\infty} x^i \\ &= \frac{x}{1-x} b(k; n, p) \\ &= \frac{kq}{np-k} b(k; n, p). \end{aligned}$$

□

Corollaire C.5 *Considérons une suite de n épreuves de Bernoulli ayant chacune une probabilité de succès p et une probabilité d'échec $q = 1 - p$. Alors, pour $0 < k \leq np/2$, la probabilité d'avoir moins de k succès est inférieure à la moitié de la probabilité d'avoir moins de $k + 1$ succès.*

Démonstration : Comme $k \leq np/2$, on a

$$\begin{aligned} \frac{kq}{np-k} &\leq \frac{(np/2)q}{np - (np/2)} \\ &= \frac{(np/2)q}{np/2} \\ &\leq 1, \end{aligned}$$

vu que $q \leq 1$. Soit X la variable aléatoire représentant le nombre de succès. Le théorème C.4 implique que la probabilité d'avoir moins de k succès est

$$\Pr \{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p).$$

On a donc

$$\begin{aligned} \frac{\Pr \{X < k\}}{\Pr \{X < k + 1\}} &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^k b(i; n, p)} \\ &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^{k-1} b(i; n, p) + b(k; n, p)} \\ &< 1/2, \end{aligned}$$

car $\sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p)$. □

On peut déterminer d'une manière similaire des bornes pour la queue de droite. Les démonstrations sont laissées en exercice (voir exercice C.5.2).

Corollaire C.6 *Considérons une suite de n épreuves de Bernoulli ayant chacune une probabilité de succès p . Soit X la variable aléatoire représentant le nombre total de*

succès. Alors, pour $np < k < n$, la probabilité d'avoir plus de k succès

$$\begin{aligned} \Pr \{X > k\} &= \sum_{i=k+1}^n b(i; n, p) \\ &< \frac{(n-k)p}{k-np} b(k; n, p). \end{aligned}$$

Corollaire C.7 *Considérons une suite de n épreuves de Bernoulli ayant chacune une probabilité de succès p et une probabilité d'échec $q = 1 - p$. Alors, pour $(np + n)/2 < k < n$, la probabilité d'avoir plus de k succès est inférieure à la moitié de la probabilité d'avoir plus de $k - 1$ succès.*

Le théorème suivant considère n épreuves de Bernoulli, chacune ayant la probabilité p_i de succès ($i = 1, 2, \dots, n$). Comme le montre le corollaire qui suit, ce théorème permet de fournir une borne pour la queue de droite de la distribution binomiale si l'on fait $p_i = p$ pour chaque épreuve.

Théorème C.8 *Soit une suite de n épreuves de Bernoulli dans laquelle la i ème épreuve ($i = 1, 2, \dots, n$) a une probabilité de succès p_i et une probabilité d'échec $q_i = 1 - p_i$. Soit X la variable aléatoire décrivant le nombre total de succès et soit $\mu = E[X]$. Alors, pour $r > \mu$, on a*

$$\Pr \{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r.$$

Démonstration : Comme, pour tout $\alpha > 0$, la fonction $e^{\alpha x}$ est strictement croissante en x ,

$$\Pr \{X - \mu \geq r\} = \Pr \{e^{\alpha(X-\mu)} \geq e^{\alpha r}\}, \quad (\text{C.41})$$

α sera déterminé ultérieurement. L'inégalité de Markov (C.29) permet d'écrire

$$\Pr \{e^{\alpha(X-\mu)} \geq e^{\alpha r}\} \leq E[e^{\alpha(X-\mu)}] e^{-\alpha r}. \quad (\text{C.42})$$

L'essentiel de la démonstration consiste à borner $E[e^{\alpha(X-\mu)}]$ et à substituer une valeur idoine à α dans l'inégalité (C.42). Primo, on évalue $E[e^{\alpha(X-\mu)}]$. En employant la notation de la section 5.2, soit $X_i = I$ {la i ème épreuve de Bernoulli est un succès} pour $i = 1, 2, \dots, n$; c'est-à-dire que X_i est la variable aléatoire qui vaut 1 si la i ème épreuve de Bernoulli est un succès et qui vaut 0 si c'est un échec. Donc

$$X = \sum_{i=1}^n X_i,$$

et, d'après la linéarité de l'espérance,

$$\mu = E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n p_i,$$

ce qui implique

$$X - \mu = \sum_{i=1}^n (X_i - p_i).$$

Pour évaluer $E[e^{\alpha(X-\mu)}]$, on substitue pour $X - \mu$, ce qui donne

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &= E\left[e^{\alpha \sum_{i=1}^n (X_i - p_i)}\right] \\ &= E\left[\prod_{i=1}^n e^{\alpha(X_i - p_i)}\right] \\ &= \prod_{i=1}^n E[e^{\alpha(X_i - p_i)}], \end{aligned}$$

qui découle de (C.23), car l'indépendance mutuelle des variables aléatoires X_i implique l'indépendance mutuelle des variables aléatoires $e^{\alpha(X_i - p_i)}$ (voir exercice C.3.5). De par la définition de l'espérance

$$\begin{aligned} E[e^{\alpha(X_i - p_i)}] &= e^{\alpha(1-p_i)}p_i + e^{\alpha(0-p_i)}q_i \\ &= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\ &\leq p_i e^{\alpha} + 1 \\ &\leq \exp(p_i e^{\alpha}), \end{aligned} \tag{C.43}$$

où $\exp(x)$ désigne la fonction exponentielle : $\exp(x) = e^x$. (L'inégalité (C.43) découle des inégalités $\alpha > 0$, $q_i \leq 1$, $e^{\alpha q_i} \leq e^{\alpha}$ et $e^{-\alpha p_i} \leq 1$, et la dernière ligne découle de l'inégalité (3.11)). Par conséquent,

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &= \prod_{i=1}^n E[e^{\alpha(X_i - p_i)}] \\ &\leq \prod_{i=1}^n \exp(p_i e^{\alpha}) \\ &= \exp\left(\sum_{i=1}^n p_i e^{\alpha}\right) \\ &= \exp(\mu e^{\alpha}), \end{aligned} \tag{C.44}$$

vu que $\mu = \sum_{i=1}^n p_i$. Donc, d'après l'équation (C.41) et les inégalités (C.42) et (C.44), il s'ensuit que

$$\Pr\{X - \mu \geq r\} \leq \exp(\mu e^{\alpha} - \alpha r). \tag{C.45}$$

En prenant $\alpha = \ln(r/\mu)$ (voir exercice C.5.7), on obtient

$$\begin{aligned} \Pr\{X - \mu \geq r\} &\leq \exp(\mu e^{\ln(r/\mu)} - r \ln(r/\mu)) \\ &= \exp(r - r \ln(r/\mu)) \\ &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r. \end{aligned}$$

□

Appliqué aux épreuves de Bernoulli dans lesquelles chaque épreuve a la même probabilité de succès, le théorème C.8 donne le majorant suivant pour la queue de droite d'une distribution binomiale.

Corollaire C.9 Soit une suite de n épreuves de Bernoulli, où chaque épreuve a une probabilité de succès p et une probabilité d'échec $q = 1 - p$. Alors, pour $r > np$,

$$\begin{aligned} \Pr \{X - np \geq r\} &= \sum_{k=\lceil np+r \rceil}^n b(k; n, p) \\ &\leq \left(\frac{npq}{r}\right)^r. \end{aligned}$$

Démonstration : D'après l'équation (C.36), on a $\mu = E[X] = np$. □

Exercices

C.5.1 * Quel événement a le moins de chance de se produire : n'obtenir aucune face quand on lance une pièce non truquée n fois, ou obtenir moins de n faces quand on lance la pièce $4n$ fois ?

C.5.2 * Démontrer les corollaires C.6 et C.7.

C.5.3 * Montrer que

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

pour tout $a > 0$ et tout k tel que $0 < k < n$.

C.5.4 * Prouver que, si $0 < k < np$, avec $0 < p < 1$ et $q = 1 - p$, alors

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np - k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

C.5.5 * Montrer que les conditions du théorème C.8 impliquent que

$$\Pr \{\mu - X \geq r\} \leq \left(\frac{(n - \mu)e}{r}\right)^r.$$

De même, montrer que les conditions du corollaire C.9 impliquent que

$$\Pr \{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r.$$

C.5.6 * On considère une séquence de n épreuves de Bernoulli où, dans la i ème épreuve ($i = 1, 2, \dots, n$), le succès est obtenu avec une probabilité p_i et l'échec avec une probabilité $q_i = 1 - p_i$. Soit X la variable aléatoire décrivant le nombre total de succès, et soit $\mu = E[X]$. Montrer que, pour $r \geq 0$,

$$\Pr \{X - \mu \geq r\} \leq e^{-r^2/2n}.$$

(Conseil : Démontrer que $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2}$. Ensuite, suivre les grandes lignes de la démonstration du théorème C.8 en se servant de cette inégalité à la place de l'inégalité (C.43).)

C.5.7 * Montrer que choisir $\alpha = \ln(r/\mu)$ minimise la partie droite de l'inégalité (C.45).

PROBLÈMES

C.1. Ballons et paniers

Dans ce problème, on cherche à connaître l'effet de diverses hypothèses sur le nombre de façons de placer n ballons dans b paniers distincts.

- On suppose que les n ballons sont distincts et que leur ordre à l'intérieur d'un panier n'importe pas. Prouver que le nombre de façons de placer les ballons dans les paniers est b^n .
- On suppose que les ballons sont distincts et que les ballons de chaque panier sont ordonnés. Démontrer que le nombre de manières de placer les ballons dans les paniers est exactement $(b + n - 1)! / (b - 1)!$ (conseil : Considérer le nombre de façons d'arranger n ballons distincts et $b - 1$ cannes identiques à la file.)
- On suppose que les ballons sont identiques, et donc que leur ordre à l'intérieur d'un panier est sans importance. Montrer que le nombre de manières de placer les ballons dans les paniers est $\binom{b+n-1}{n}$. (conseil : Parmi les arrangements de la partie (b), combien sont répétés si les ballons sont rendus identiques ?)
- On suppose que les ballons sont identiques et qu'aucun panier ne peut contenir plus d'un seul ballon. Montrer que le nombre de façons de placer les ballons est $\binom{b}{n}$.
- On suppose que les ballons sont identiques et qu'aucun panier ne peut être laissé vide. Montrer que le nombre de façons de placer les ballons est $\binom{n-1}{b-1}$.

NOTES

Les premières méthodes générales de résolution des problèmes de probabilités ont été étudiées dans une correspondance fameuse entre B. Pascal et P. de Fermat, qui débuta en 1654, ainsi que dans un livre de C. Huygens en 1657. Une théorie rigoureuse des probabilités trouva son origine dans les travaux de J. Bernoulli en 1713 et de A. de Moivre en 1730. D'autres développements furent apportés à la théorie par P. S. de Laplace, S.-D. Poisson et C. F. Gauss.

Les sommes de variables aléatoires ont d'abord été étudiées par P. L. Chebyshev et A. A. Markov. La théorie des probabilités fut axiomatisée par A. N. Kolmogorov en 1933. Des bornes sur les queues de distribution ont été données par Chernoff [59] et Hoeffding [150]. Le premier travail sur les structures combinatoires aléatoires fut effectué par P. Erdős.

Knuth [182] et Liu [205] sont de bonnes références pour les bases de la combinatoire et du dénombrement. Des manuels classiques comme (en français) Guessarian et Arnold [325] ou (en américain) Billingsley [42], Chung [60], Drake [80], Feller [88] et Rozanov [263] offrent une introduction complète aux probabilités.

Bibliographie

- [1] Abramowitz (Milton) and Stegun (Irene A). editors. *Handbook of Mathematical Functions*. Dover, 1965.
- [2] Adel'son-Vel'skiĭ (G. M.) and Landis (E. M.). An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3 : 1259–1263, 1962.
- [3] Adleman (Leonard M.), Pomerance (Carl), and Rumely (Robert S.). On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117 : 173–206, 1983.
- [4] Aggarwal (Alok) and Vitter (Jeffrey Scott). The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9) : 1116–1127, 1988.
- [5] Aho (Alfred V.), Hopcroft (John E.), and Ullman (Jeffrey D.). *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [6] Aho (Alfred V.), Hopcroft (John E.), and Ullman (Jeffrey D.). *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [7] Ahuja (Ravindra K.), Magnanti (Thomas L.), and Orlin (James B.). *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [8] Ahuja (Ravindra K.), Mehlhorn (Kurt), Orlin (James B.), and Tarjan (Robert E.). Faster algorithms for the shortest path problem. *Journal of the ACM*, 37 : 213–223, 1990.
- [9] Ahuja (Ravindra K.) and Orlin (James B.). A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5) : 748–759, 1989.

- [10] Ahuja (Ravindra K.), Orlin (James B.), and Tarjan (Robert E.). Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5) : 939–954, 1989.
- [11] Ajtai (Miklos), Komlós (János), and Szemerédi (Endre). Sorting in $c \log n$ parallel steps. *Combinatorica*, 3 : 1–19, 1983.
- [12] Ajtai (Miklos), Megiddo (Nimrod), and Waarts (Orli). Improved algorithms and analysis for secretary problems and generalizations. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 473–482, 1995.
- [13] Akra (Mohamad) and Bazzi (Louay). On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2) : 195–210, 1998.
- [14] Alon (Noga). Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35 : 201–204, 1990.
- [15] Andersson (Arne). Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, number 709 in Lecture Notes in Computer Science, pages 60–71. Springer-Verlag, 1993.
- [16] Andersson (Arne). Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 135–141, 1996.
- [17] Andersson (Arne), Hagerup (Torben), Nilsson (Stefan), and Raman (Rajeev). Sorting in linear time? *Journal of Computer and System Sciences*, 57 : 74–93, 1998.
- [18] Apostol (Tom M.). *Calculus*, volume 1. Blaisdell Publishing Company, second edition, 1967.
- [19] Arora (Sanjeev). *Probabilistic checking of proofs and the hardness of approximation problems*. PhD thesis, University of California, Berkeley, 1994.
- [20] Arora (Sanjeev). The approximability of NP-hard problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 337–348, 1998.
- [21] Arora (Sanjeev). Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5) : 753–782, 1998.
- [22] Arora (Sanjeev) and Lund (Carsten). Hardness of approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 399–446. PWS Publishing Company, 1997.

- [23] Aslam (Javed A.). A simple bound on the expected height of a randomly built binary search tree. Technical Report TR2001-387, Dartmouth College Department of Computer Science, 2001.
- [24] Atallah (Mikhail J.), editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
- [25] Ausiello (G.), Crescenzi (P.), Gambosi (G.), Kann (V.), Marchetti-Spaccamela (A.), and Protasi (M.). *Complexity and Approximation : Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999.
- [26] Baase (Sara) and Van Gelder (Alan). *Computer Algorithms : Introduction to Design and Analysis*. Addison-Wesley, third edition, 2000.
- [27] Bach (Eric). Private communication, 1989.
- [28] Bach (Eric). Number-theoretic algorithms. In *Annual Review of Computer Science*, volume 4, pages 119–172. Annual Reviews, Inc., 1990.
- [29] Bach (Eric) and Shallit (Jeffery). *Algorithmic Number Theory—Volume I : Efficient Algorithms*. The MIT Press, 1996.
- [30] Bailey (David H.), Lee (King), and Simon (Horst D.). Using Strassen’s algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4) : 357–371, 1990.
- [31] Bayer (R.). Symmetric binary B-trees : Data structure and maintenance algorithms. *Acta Informatica*, 1 : 290–306, 1972.
- [32] Bayer (R.) and McCreight (E. M.). Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3) : 173–189, 1972.
- [33] Beauchemin (Pierre), Brassard (Gilles), Crépeau (Claude), Goutier (Claude), and Pomerance (Carl). The generation of random numbers that are probably prime. *Journal of Cryptology*, 1 : 53–64, 1988.
- [34] Bellman (Richard). *Dynamic Programming*. Princeton University Press, 1957.
- [35] Bellman (Richard). On a routing problem. *Quarterly of Applied Mathematics*, 16(1) : 87–90, 1958.
- [36] Ben-Or (Michael). Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.
- [37] Bender (Michael A.), Demaine (Erik D.), and Farach-Colton (Martin). Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

- [38] Bent (Samuel W.) and John (John W.). Finding the median requires $2n$ comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985.
- [39] Bentley (Jon L.). *Writing Efficient Programs*. Prentice-Hall, 1982.
- [40] Bentley (Jon L.). *Programming Pearls*. Addison-Wesley, 1986.
- [41] Bentley (Jon L.), Haken (Dorothea), and Saxe (James B.). A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3) : 36–44, 1980.
- [42] Billingsley (Patrick). *Probability and Measure*. John Wiley & Sons, second edition, 1986.
- [43] Blum (Manuel), Floyd (Robert W.), Pratt (Vaughan), Rivest (Ronald L.), and Tarjan (Robert E.). Time bounds for selection. *Journal of Computer and System Sciences*, 7(4) : 448–461, 1973.
- [44] Bollobás (Béla). *Random Graphs*. Academic Press, 1985.
- [45] Bondy (J. A.) and Murty (U. S. R.). *Graph Theory with Applications*. American Elsevier, 1976.
- [46] Brassard (Gilles) and Bratley (Paul). *Algorithmics : Theory and Practice*. Prentice-Hall, 1988.
- [47] Brassard (Gilles) and Bratley (Paul). *Fundamentals of Algorithmics*. Prentice Hall, 1996.
- [48] Brent (Richard P.). An improved Monte Carlo factorization algorithm. *BIT*, 20(2) : 176–184, 1980.
- [49] Brown (Mark R.). *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Computer Science Department, Stanford University, 1977. Technical Report STAN-CS-77-600.
- [50] Brown (Mark R.). Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3) : 298–319, 1978.
- [51] Buhler (J. P.), Lenstra (H. W.), Jr., and Pomerance (Carl). Factoring integers with the number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 50–94. Springer-Verlag, 1993.
- [52] Carter (J. Lawrence) and Wegman (Mark N.). Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2) : 143–154, 1979.
- [53] Chazelle (Bernard). A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6) : 1028–1047, 2000.

- [54] Cheriyan (Joseph) and Hagerup (Torben). A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2) : 203–226, 1995.
- [55] Cheriyan (Joseph) and Maheshwari (S. N.). Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6) : 1057–1086, 1989.
- [56] Cherkassky (Boris V.) and Goldberg (Andrew V.). On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4) : 390–410, 1997.
- [57] Cherkassky (Boris V.), Goldberg (Andrew V.), and Radzik (Tomasz). Shortest paths algorithms : Theory and experimental evaluation. *Mathematical Programming*, 73(2) : 129–174, 1996.
- [58] Cherkassky (Boris V.), Goldberg (Andrew V.), and Silverstein (Craig). Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4) : 1326–1346, 1999.
- [59] Chernoff (H.). A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23 : 493–507, 1952.
- [60] Chung (Kai Lai). *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974.
- [61] Chvátal (V.). A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3) : 233–235, 1979.
- [62] Chvátal (V.). *Linear Programming*. W. H. Freeman and Company, 1983.
- [63] Chvátal (V.), Klarner (D. A.), and Knuth (D. E.). Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.
- [64] Cobham (Alan). The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.
- [65] Cohen (H.) and Lenstra (H. W.), Jr. Primality testing and Jacobi sums. *Mathematics of Computation*, 42(165) : 297–330, 1984.
- [66] Comer (D.). The ubiquitous B-tree. *ACM Computing Surveys*, 11(2) : 121–137, 1979.
- [67] Cook (Stephen). The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

- [68] Cooley (James W.) and Tukey (John W.). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90) : 297–301, 1965.
- [69] Coppersmith (Don). Modifications to the number field sieve. *Journal of Cryptology*, 6 : 169–180, 1993.
- [70] Coppersmith (Don) and Winograd (Shmuel). Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3) : 251–280, 1990.
- [71] Cormen (Thomas H.). *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.
- [72] Denardo (Eric V.) and Fox (Bennett L.). Shortest-route methods : 1. Reaching, pruning, and buckets. *Operations Research*, 27(1) : 161–186, 1979.
- [73] Dietzfelbinger (Martin), Karlin (Anna), Mehlhorn (Kurt), Meyer auf der Heide (Friedhelm), Rohnert (Hans), and Tarjan (Robert E.). Dynamic perfect hashing : Upper and lower bounds. *SIAM Journal on Computing*, 23(4) : 738–761, 1994.
- [74] Diffie (Whitfield) and Hellman (Martin E.). New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6) : 644–654, 1976.
- [75] Dijkstra (E. W.). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 : 269–271, 1959.
- [76] Dinic (E. A.). Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5) : 1277–1280, 1970.
- [77] Dixon (Brandon), Rauch (Monika), and Tarjan (Robert E.). Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6) : 1184–1192, 1992.
- [78] Dixon (John D.). Factorization and primality tests. *The American Mathematical Monthly*, 91(6) : 333–352, 1984.
- [79] Dor (Dorit) and Zwick (Uri). Selecting the median. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 28–37, 1995.
- [80] Drake (Alvin W.). *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.
- [81] Driscoll (James R.), Gabow (Harold N.), Shrairman (Ruth), and Tarjan (Robert E.). Relaxed heaps : An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11) : 1343–1354, 1988.

- [82] Driscoll (James R.), Sarnak (Neil), Sleator (Daniel D.), and Tarjan (Robert E.). Making data structures persistent. *Journal of Computer and System Sciences*, 38 : 86–124, 1989.
- [83] Edelsbrunner (Herbert). *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.
- [84] Edmonds (Jack). Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17 : 449–467, 1965.
- [85] Edmonds (Jack). Matroids and the greedy algorithm. *Mathematical Programming*, 1 : 126–136, 1971.
- [86] Edmonds (Jack) and Karp (Richard M.). Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19 : 248–264, 1972.
- [87] Even (Shimon). *Graph Algorithms*. Computer Science Press, 1979.
- [88] Feller (William). *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.
- [89] Floyd (Robert W.). Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6) : 345, 1962.
- [90] Floyd (Robert W.). Algorithm 245 (TREESORT). *Communications of the ACM*, 7 : 701, 1964.
- [91] Floyd (Robert W.). Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, 1972.
- [92] Floyd (Robert W.) and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3) : 165–172, 1975.
- [93] Ford (Lester R.), Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- [94] Ford (Lester R.), Jr. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66 : 387–389, 1959.
- [95] Fredman (Michael L.). New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1) : 83–89, 1976.
- [96] Fredman (Michael L.), Komlós (János), and Szermerédi (Endre). Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3) : 538–544, 1984.

- [97] Fredman (Michael L.) and Saks (Michael E.). The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.
- [98] Fredman (Michael L.) and Tarjan (Robert E.). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3) : 596–615, 1987.
- [99] Fredman (Michael L.) and Willard (Dan E.). Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47 : 424–436, 1993.
- [100] Fredman (Michael L.) and Willard (Dan E.). Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3) : 533–551, 1994.
- [101] Gabow (Harold N.). Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74 : 107–114, 2000.
- [102] Gabow (Harold N.), Galil (Z.), Spencer (T.), and Tarjan (Robert E.). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2) : 109–122, 1986.
- [103] Gabow (Harold N.) and Tarjan (Robert E.). A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2) : 209–221, 1985.
- [104] Gabow (Harold N.) and Tarjan (Robert E.). Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5) : 1013–1036, 1989.
- [105] Galil (Zvi) and Margalit (Oded). All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2) : 103–139, 1997.
- [106] Galil (Zvi) and Margalit (Oded). All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2) : 243–254, 1997.
- [107] Galil (Zvi) and Seiferas (Joel). Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3) : 280–294, 1983.
- [108] Galperin (Igal) and Rivest (Ronald L.). Scapegoat trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.
- [109] Garey (Michael R.), Graham (R. L.), and Ullman (J. D.). Worst-case analysis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pages 143–150, 1972.

- [110] Garey (Michael R.) and Johnson (David S.). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [111] Gass (Saul). *Linear Programming : Methods and Applications*. International Thomson Publishing, fourth edition, 1975.
- [112] Gavril (Fănică). Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2) : 180–187, 1972.
- [113] George (Alan) and Liu (Joseph W-H). *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.
- [114] Gilbert (E. N.) and Moore (E. F.). Variable-length binary encodings. *Bell System Technical Journal*, 38(4) : 933–967, 1959.
- [115] Goemans (Michel X.) and Williamson (David P.). Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6) : 1115–1145, 1995.
- [116] Goemans (Michel X.) and Williamson (David P.). The primal-dual method for approximation algorithms and its application to network design problems. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 144–191. PWS Publishing Company, 1997.
- [117] Goldberg (Andrew V.). *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.
- [118] Goldberg (Andrew V.). Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3) : 494–504, 1995.
- [119] Goldberg (Andrew V.), Tardos (Éva), and Tarjan (Robert E.). Network flow algorithms. In Bernhard Korte, László Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI-Layout*, pages 101–164. Springer-Verlag, 1990.
- [120] Goldberg (Andrew V.) and Rao (Satish). Beyond the flow decomposition barrier. *Journal of the ACM*, 45 : 783–797, 1998.
- [121] Goldberg (Andrew V.) and Tarjan (Robert E.). A new approach to the maximum flow problem. *Journal of the ACM*, 35 : 921–940, 1988.
- [122] Goldfarb (D.) and Todd (M. J.). Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy-Kan, and M. J. Todd, editors, *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, pages 73–170. Elsevier Science Publishers, 1989.

- [123] Goldwasser (Shafi) and Micali (Silvio). Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2) : 270–299, 1984.
- [124] Goldwasser (Shafi), Micali (Silvio), and Rivest (Ronald L.). A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2) : 281–308, 1988.
- [125] Golub (Gene H.) and Van Loan (Charles F.). *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.
- [126] Gonnet (G. H.). *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.
- [127] Gonzalez (Rafael C.) and Woods (Richard E.). *Digital Image Processing*. Addison-Wesley, 1992.
- [128] Goodrich (Michael T.) and Tamassia (Roberto). *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.
- [129] Graham (Ronald L.). Bounds for certain multiprocessor anomalies. *Bell Systems Technical Journal*, 45 : 1563–1581, 1966.
- [130] Graham (Ronald L.). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1 : 132–133, 1972.
- [131] Graham (Ronald L.) and Hell (Pavol). On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1) : 43–57, 1985.
- [132] Graham (Ronald L.), Knuth (Donald E.), and Patashnik (Oren). *Concrete Mathematics*. Addison-Wesley, second edition, 1994.
- [133] Gries (David). *The Science of Programming*. Springer-Verlag, 1981.
- [134] Grötschel (M.), Lovász (László), and Schrijver (Alexander). *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.
- [135] Guibas (Leo J.) and Sedgewick (Robert). A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.
- [136] Gusfield (Dan). *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [137] Han (Yijie). Improved fast integer sorting in linear space. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 793–796, 2001.
- [138] Harary (Frank). *Graph Theory*. Addison-Wesley, 1969.
- [139] Harfst (Gregory C.) and M. Reingold (Edward). A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31(3) : 86–95, 2000.

- [140] Hartmanis (J.) and Stearns (R. E.). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117 : 285–306, 1965.
- [141] Heideman (Michael T.), Johnson (Don H.), and Burrus (C. Sidney). Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, pages 14–21, 1984.
- [142] Henzinger (Monika R.) and King (Valerie). Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 664–672, 1995.
- [143] Henzinger (Monika R.) and King (Valerie). Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4) : 502–516, 1999.
- [144] Henzinger (Monika R.), Rao (Satish), and Gabow (Harold N.). Computing vertex connectivity : New bounds from old techniques. *Journal of Algorithms*, 34(2) : 222–250, 2000.
- [145] Higham (Nicholas J.). Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4) : 352–368, 1990.
- [146] Hoare (C. A. R.). Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7) : 321–322, 1961.
- [147] Hoare (C. A. R.). Quicksort. *Computer Journal*, 5(1) : 10–15, 1962.
- [148] Hochbaum (Dorit S.). Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Math*, 6 : 243–254, 1983.
- [149] Hochbaum (Dorit S.), editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.
- [150] Hoeffding (W.). On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27 : 713–721, 1956.
- [151] Hofri (Micha). *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987.
- [152] Hopcroft (John E.) and Karp (Richard M.). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4) : 225–231, 1973.
- [153] Hopcroft (John E.), Motwani (Rajeev), and Ullman (Jeffrey D.). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.
- [154] Hopcroft (John E.) and Tarjan (Robert E.). Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6) : 372–378, 1973.

- [155] Hopcroft (John E.) and Ullman (Jeffrey D.). Set merging algorithms. *SIAM Journal on Computing*, 2(4) : 294–303, 1973.
- [156] Hopcroft (John E.) and Ullman (Jeffrey D.). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [157] Horowitz (Ellis) and Sahni (Sartaj). *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
- [158] Horowitz (Ellis), Sahni (Sartaj), and Rajasekaran (Sanguthevar). *Computer Algorithms*. Computer Science Press, 1998.
- [159] Hu (T. C.) and Shing (M. T.). Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2) : 362–373, 1982.
- [160] Hu (T. C.) and Shing (M. T.). Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2) : 228–251, 1984.
- [161] Hu (T. C.) and Tucker (A. C.). Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4) : 514–532, 1971.
- [162] Huffman (David A.). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9) : 1098–1101, 1952.
- [163] Huss-Lederman (Steven), Jacobson (Elaine M.), Johnson (Jeremy R.), Tsao (Anna), and Turnbull (Thomas). Implementation of Strassen’s algorithm for matrix multiplication. In *SC96 Technical Papers*, 1996.
- [164] Ibarra (Oscar H.) and Kim (Chul E.). Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4) : 463–468, 1975.
- [165] Jarvis (R. A.). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2 : 18–21, 1973.
- [166] Johnson (David S.). Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9 : 256–278, 1974.
- [167] Johnson (David S.). The NP-completeness column : An ongoing guide—the tale of the second prover. *Journal of Algorithms*, 13(3) : 502–524, 1992.
- [168] Johnson (Donald B.). Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1) : 1–13, 1977.
- [169] Karger (David R.), Klein (Philip N.), and Tarjan (Robert E.). A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2) : 321–328, 1995.

- [170] Karger (David R.), Koller (Daphne), and Phillips (Steven J.). Finding the hidden path : time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6) : 1199–1217, 1993.
- [171] Karloff (Howard). *Linear Programming*. Birkäuser, 1991.
- [172] Karmarkar (N.). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4) : 373–395, 1984.
- [173] Karp (Richard M.). Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [174] Karp (Richard M.). An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34 : 165–201, 1991.
- [175] Karp (Richard M.) and Rabin (Michael O.). Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.
- [176] Karzanov (A. V.). Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15 : 434–437, 1974.
- [177] King (Valerie). A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2) : 263–270, 1997.
- [178] King (Valerie), Rao (Satish), and Tarjan (Robert E.). A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17 : 447–474, 1994.
- [179] Kingston (Jeffrey H.). *Algorithms and Data Structures : Design, Correctness, Analysis*. Addison-Wesley, 1990.
- [180] Kirkpatrick (D. G.) and Seidel (R.). The ultimate planar convex hull algorithm ? *SIAM Journal on Computing*, 15(2) : 287–299, 1986.
- [181] Klein (Philip N.) and Young (Neal E.). Approximation algorithms for NP-hard optimization problems. In *CRC Handbook on Algorithms*, pages 34-1–34-19. CRC Press, 1999.
- [182] Knuth (Donald E.). *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.
- [183] Knuth (Donald E.). *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969. Second edition, 1981.
- [184] Knuth (Donald E.). Optimum binary search trees. *Acta Informatica*, 1 : 14–25, 1971.
- [185] Knuth (Donald E.). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

- [186] Knuth (Donald E.). Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2) : 18–23, 1976.
- [187] Knuth (Donald E.), James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2) : 323–350, 1977.
- [188] Komlós (J.). Linear verification for spanning trees. *Combinatorica*, 5(1) : 57–65, 1985.
- [189] Korte (Bernhard) and Lovász (László). Mathematical structures underlying greedy algorithms. In F. Geceg, editor, *Fundamentals of Computation Theory*, number 117 in Lecture Notes in Computer Science, pages 205–209. Springer-Verlag, 1981.
- [190] Korte (Bernhard) and Lovász (László). Structural properties of greedoids. *Combinatorica*, 3 : 359–374, 1983.
- [191] Korte (Bernhard) and Lovász (László). Greedoids—a structural framework for the greedy algorithm. In W. Pulleybank, editor, *Progress in Combinatorial Optimization*, pages 221–243. Academic Press, 1984.
- [192] Korte (Bernhard) and Lovász (László). Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2) : 229–238, 1984.
- [193] Kozen (Dexter C.). *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.
- [194] Krumme (David W.), Cybenko (George), and Venkataraman (K. N.). Gossiping in minimal time. *SIAM Journal on Computing*, 21(1) : 111–139, 1992.
- [195] Kruskal (J. B.). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7 : 48–50, 1956.
- [196] Lawler (Eugene L.). *Combinatorial Optimization : Networks and Matroids*. Holt, Rinehart, and Winston, 1976.
- [197] Lawler (Eugene L.), Lenstra (J. K.), Rinnooy Kan (A. H. G.), and Shmoys (D. B.), editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.
- [198] Lee (C. Y.). An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3) : 346–365, 1961.
- [199] Leighton (Tom) and Rao (Satish). An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

- [200] Lelewer (Debra A.) and Hirschberg (Daniel S). Data compression. *ACM Computing Surveys*, 19(3) : 261–296, 1987.
- [201] Lenstra (A. K.), Lenstra (H. W.), Jr., Manasse (M. S.), and Pollard (J. M.). The number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42. Springer-Verlag, 1993.
- [202] Lenstra (H. W.), Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126 : 649–673, 1987.
- [203] Levin (L. A.). Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3) : 265–266, 1973. In Russian.
- [204] Lewis (Harry R.) and H. (Christos) Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1998.
- [205] Liu (C. L.). *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.
- [206] Lovász (László). On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13 : 383–390, 1975.
- [207] Lovász (László) and Plummer (M. D.). *Matching Theory*, volume 121 of *Annals of Discrete Mathematics*. North Holland, 1986.
- [208] Maggs (Bruce M.) and Plotkin (Serge A.). Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6) : 291–293, 1988.
- [209] Main (Michael). *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.
- [210] Manber (Udi). *Introduction to Algorithms : A Creative Approach*. Addison-Wesley, 1989.
- [211] Martínez (Conrado) and Roura (Salvador). Randomized binary search trees. *Journal of the ACM*, 45(2) : 288–323, 1998.
- [212] Masek (William J.) and Paterson (Michael S.). A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1) : 18–31, 1980.
- [213] Maurer (H. A.), Ottmann (Th.), and Six (H.-W.). Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1) : 11–14, 1976.
- [214] Mayr (Ernst W.), Prömel (Hans Jürgen), and Steger (Angelika), editors. *Lectures on Proof Verification and Approximation Algorithms*. Number 1367 in *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [215] McGeoch (C. C.). All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5) : 426–441, 1995.

- [216] McIlroy (M. D.). A killer adversary for quicksort. *Software—Practice and Experience*, 29(4) : 341–344, 1999.
- [217] Mehlhorn (Kurt). *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
- [218] Mehlhorn (Kurt). *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
- [219] Mehlhorn (Kurt). *Multidimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
- [220] Menezes (Alfred J.), van Oorschot (Paul C.), and Vanstone (Scott A.). *Handbook of Applied Cryptography*. CRC Press, 1997.
- [221] Miller (Gary L.). Riemann’s hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3) : 300–317, 1976.
- [222] Mitchell (John C.). *Foundations for Programming Languages*. MIT Press, 1996.
- [223] Mitchell (Joseph S. B.). Guillotine subdivisions approximate polygonal subdivisions : A simple polynomial-time approximation scheme for geometric TSP, k -MST, and related problems. *SIAM Journal on Computing*, 28(4) : 1298–1309, 1999.
- [224] Monier (Louis). *Algorithmes de Factorisation d’Entiers*. Centre d’Orsay, Thèse de doctorat, Université Paris-Sud, 1980.
- [225] Monier (Louis). Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1) : 97–108, 1980.
- [226] Moore (Edward F.). The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.
- [227] Motwani (Rajeev), Naor (Joseph (Seffi)), and Raghavan (Prabakar). Randomized approximation algorithms in combinatorial optimization. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 447–481. PWS Publishing Company, 1997.
- [228] Motwani (Rajeev) and Raghavan (Prabhakar). *Randomized Algorithms*. Cambridge Unveristy Press, 1995.
- [229] Munro (J. I.) and Raman (V.). Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2) : 151–160, 1996.
- [230] Nievergelt (J.) and Reingold (E. M.). Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1) : 33–43, 1973.

- [231] Niven (Ivan) and Zuckerman (Herbert S.). *An Introduction to the Theory of Numbers*. John Wiley & Sons, fourth edition, 1980.
- [232] Oppenheim (Alan V.) and Schaffer (Ronald W.), with John R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, second edition, 1998.
- [233] Oppenheim (Alan V.) and Willsky (Alan S.), with S. Nawab (Hamid). *Signals and Systems*. Prentice-Hall, second edition, 1997.
- [234] Orlin (James B.). A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1) : 109–129, 1997.
- [235] O'Rourke (Joseph). *Computational Geometry in C*. Cambridge University Press, 1993.
- [236] Papadimitriou (Christos H.). *Computational Complexity*. Addison-Wesley, 1994.
- [237] Papadimitriou (Christos H.) and Steiglitz (Kenneth). *Combinatorial Optimization : Algorithms and Complexity*. Prentice-Hall, 1982.
- [238] Paterson (Michael S.), 1974. Unpublished lecture, Ile de Berder, France.
- [239] Paterson (Michael S.). Progress in selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pages 368–379, 1996.
- [240] Pevzner (Pavel A.). *Computational Molecular Biology : An Algorithmic Approach*. The MIT Press, 2000.
- [241] Phillips (Steven) and Westbrook (Jeffery). Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.
- [242] Pollard (J. M.). A Monte Carlo method for factorization. *BIT*, 15 : 331–334, 1975.
- [243] Pollard (J. M.). Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10. Springer, 1993.
- [244] Pomerance Carl). On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156) : 587–593, 1981.
- [245] Pomerance Carl), editor. *Proceedings of the AMS Symposia in Applied Mathematics : Computational Number Theory and Cryptography*. American Mathematical Society, 1990.
- [246] Pratt William K.). *Digital Image Processing*. John Wiley & Sons, second edition, 1991.

- [247] Preparata (Franco P.) and Shamos (Michael Ian). *Computational Geometry : An Introduction*. Springer-Verlag, 1985.
- [248] Press (William H.), Flannery (Brian P.), Teukolsky Saul A.), and Vetterling (William T.). *Numerical Recipes : The Art of Scientific Computing*. Cambridge University Press, 1986.
- [249] Press (William H.), Flannery (Brian P.), Teukolsky (Saul A.), and Vetterling (William T.) . *Numerical Recipes in C*. Cambridge University Press, 1988.
- [250] Prim (R. C.). Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36 : 1389–1401, 1957.
- [251] Pugh (William). Skip lists : A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6) : 668–676, 1990.
- [252] Purdom (Paul W.), Jr. and Brown (Cynthia A.). *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.
- [253] Rabin (Michael O.). Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity : New Directions and Recent Results*, pages 21–39. Academic Press, 1976.
- [254] Rabin (Michael O.). Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1) : 128–138, 1980.
- [255] Raghavan (P.) and Thompson (C. D.). Randomized rounding : A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7 : 365–374, 1987.
- [256] Raman (Rajeev). Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2) : 81–87, 1997.
- [257] Reingold (Edward M.), Nievergelt (Jürg), and Deo (Narsingh). *Combinatorial Algorithms : Theory and Practice*. Prentice-Hall, 1977.
- [258] Riesel (Hans). *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser, 1985.
- [259] Rivest (Ronald L.), Shamir (Adi), and Adleman (Leonard M.). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) : 120–126, 1978. See also U.S. Patent 4,405,829.
- [260] Robbins (Herbert). A remark on Stirling’s formula. *American Mathematical Monthly*, 62(1) : 26–29, 1955.
- [261] Rosenkrantz (D. J.), Stearns (R. E.), and Lewis (P. M.). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6 : 563–581, 1977.

- [262] Roura (Salvador). An improved master theorem for divide-and-conquer recurrences. In *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pages 449–459, 1997.
- [263] Rozanov (Y. A.). *Probability Theory : A Concise Course*. Dover, 1969.
- [264] Sahni (S.) and Gonzalez (T.). P-complete approximation problems. *Journal of the ACM*, 23 : 555–565, 1976.
- [265] Schönhage (A.), Paterson (M.), and Pippenger (N.). Finding the median. *Journal of Computer and System Sciences*, 13(2) : 184–199, 1976.
- [266] Schrijver (Alexander). *Theory of linear and integer programming*. John Wiley & Sons, 1986.
- [267] Schrijver (Alexander). Paths and flows—a historical survey. *CWI Quarterly*, 6 : 169–183, 1993.
- [268] Sedgewick (Robert). Implementing quicksort programs. *Communications of the ACM*, 21(10) : 847–857, 1978.
- [269] Sedgewick (Robert). *Algorithms*. Addison-Wesley, second edition, 1988.
- [270] Seidel (Raimund). On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3) : 400–403, 1995.
- [271] Seidel (Raimund) and Aragon (C. R.). Randomized search trees. *Algorithmica*, 16 : 464–497, 1996.
- [272] Setubal (João) and Meidanis (João). *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
- [273] Shaffer (Clifford A.). *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, second edition, 2001.
- [274] Shallit (Jeffrey). Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4) : 401–419, 1994.
- [275] Shamos (Michael I.) and Hoey (Dan). Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 208–215, 1976.
- [276] Sharir (M.). A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7 : 67–72, 1981.
- [277] Shmoys (David B.). Computing near-optimal solutions to combinatorial optimization problems. In William Cook, László Lovász, and Paul Seymour, editors, *Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.

- [278] Shoshan (Avi) and Zwick (Uri). All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 605–614, 1999.
- [279] Sipser (Michael). *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
- [280] Skiena (Steven S.). *The Algorithm Design Manual*. Springer-Verlag, 1998.
- [281] Sleator (Daniel D.) and Tarjan (Robert E.). A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3) : 362–391, 1983.
- [282] Sleator (Daniel D.) and Tarjan (Robert E.). Self-adjusting binary search trees. *Journal of the ACM*, 32(3) : 652–686, 1985.
- [283] Spencer (Joel). *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987.
- [284] Spielman (Daniel A.) and Shang-Hua Teng. The simplex algorithm usually takes a polynomial number of steps. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001.
- [285] Strang (Gilbert). *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.
- [286] Strang (Gilbert). *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, third edition, 1988.
- [287] Strassen (Volker). Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3) : 354–356, 1969.
- [288] Szymanski (T. G.). A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.
- [289] Tarjan (Robert E.). Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) : 146–160, 1972.
- [290] Tarjan (Robert E.). Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2) : 215–225, 1975.
- [291] Tarjan (Robert E.). A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2) : 110–127, 1979.
- [292] Tarjan (Robert E.). *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.
- [293] Tarjan (Robert E.). Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2) : 306–318, 1985.

- [294] Tarjan (Robert E.). Class notes : Disjoint set union. COS 423, Princeton University, 1999.
- [295] Tarjan (Robert E.) and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2) : 245–281, 1984.
- [296] Thomas (George B.), Jr. and Ross L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, seventh edition, 1988.
- [297] Thorup (Mikkel). Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.
- [298] Thorup (Mikkel). Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3) : 362–394, 1999.
- [299] Thorup (Mikkel). On RAM priority queues. *SIAM Journal on Computing*, 30(1) : 86–109, 2000.
- [300] Tolimieri (Richard), An (Myoung), and Lu (Chao). *Mathematics of Multi-dimensional Fourier Transform Algorithms*. Springer-Verlag, second edition, 1997.
- [301] van Emde Boas (P.). Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84. IEEE Computer Society, 1975.
- [302] van Leeuwen (Jan), editor. *Handbook of Theoretical Computer Science, Volume A : Algorithms and Complexity*. Elsevier Science Publishers and The MIT Press, 1990.
- [303] Van Loan (Charles). *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.
- [304] Vanderbei (Robert J.). *Linear Programming : Foundations and Extensions*. Kluwer Academic Publishers, 1996.
- [305] Vazirani (Vijay V.). *Approximation Algorithms*. Springer-Verlag, 2001.
- [306] Verma (Rakesh M.). General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2) : 568–581, 1997.
- [307] Vuillemin (Jean). A data structure for manipulating priority queues. *Communications of the ACM*, 21(4) : 309–315, 1978.
- [308] Warshall (Stephen). A theorem on boolean matrices. *Journal of the ACM*, 9(1) : 11–12, 1962.
- [309] Waterman (Michael S.). *Introduction to Computational Biology, Maps, Sequences and Genomes*. Chapman & Hall, 1995.

- [310] Weiss (Mark Allen). *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1994.
- [311] Weiss (Mark Allen). *Algorithms, Data Structures and Problem Solving with C++*. Addison-Wesley, 1996.
- [312] Weiss (Mark Allen). *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.
- [313] Weiss (Mark Allen). *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999.
- [314] Whitney (Hassler). On the abstract properties of linear dependence. *American Journal of Mathematics*, 57 : 509–533, 1935.
- [315] Wilf (Herbert S.). *Algorithms and Complexity*. Prentice-Hall, 1986.
- [316] Williams (J. W. J.). Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7 : 347–348, 1964.
- [317] Winograd (S.). On the algebraic complexity of functions. In *Actes du Congrès International des Mathématiciens*, volume 3, pages 283–288, 1970.
- [318] Yao (Andrew C.-C.). A lower bound to finding convex hulls. *Journal of the ACM*, 28(4) : 780–787, 1981.
- [319] Ye (Yinyu). *Interior Point Algorithms : Theory and Analysis*. John Wiley & Sons, 1997.
- [320] Zwillinger (Daniel), editor. *CRC Standard Mathematical Tables and Formulae*. CRC Press, 30th edition, 1996.

Compléments bibliographiques pour l'édition française

- [321] Aho (Alfred V.), Hopcroft (John E.) et Ullman (Jeffrey D.). *Structures de données et algorithmes*, InterEditions, 1987.
- [322] Aho (Alfred V.) et Ullman (Jeffrey D.). *Concepts fondamentaux de l'informatique*, Dunod, 1993.
- [323] Berge (Claude). *Graphes et hypergraphes*, Dunod, 1970.
- [324] Froidevaux (Christine), Gaudel (Marie-Claude) et Soria (Michèle). *Types de données et algorithmes*, McGraw-Hill, 1990.
- [325] Arnold (André) et Guessarian (Irène). *Mathématiques pour l'informatique*, Masson, troisième édition 1997.

Index

- $\alpha(n)$, 499
- O , 41–43
- o , 46
- O' , 56
- \tilde{O} , 56
- Ω , 41, 43, 44
- ω , 46
- $\tilde{\Omega}$, 56
- $\tilde{\Omega}$, 41–56
- Θ , 56
- $\{ \}$ (ensemble), 1033
- \in (symbole d'appartenance), 1033
- \notin (symbole de non appartenance), 1033
- \emptyset (ensemble vide), 1034
- \subseteq (sous-ensemble), 1034
- \subset (sous-ensemble propre), 1034
- $:$ (tel que), 1034
- \cap (intersection d'ensembles), 1034
- \cup (union d'ensembles), 1034
- $-$ (différence d'ensembles), 1034
- $||$ (cardinal d'un ensemble), 1036
- \times , 903, 1037
 - produit cartésien, 1037
 - produit croisé, 903
- $\langle \rangle$ (séquence), 1041
- $\binom{n}{k}$ (combinaison), 1060
- $!$ (factorielle), 52
- $\lceil \rceil$ (partie entière supérieure), 49
- \sum (somme), 1022
- \prod (produit), 1024
- \rightsquigarrow (relation d'accessibilité), 1044
- \rightarrow (relation d'adjacence), 1044
- \wedge (ET), 614, 954
- \neg (NON), 954
- \vee (OU), 614, 954
- \oplus , 833
 - opérateur de groupe, 833
- \otimes , 798
 - opérateur de convolution, 798
- $*$ opérateur de fermeture, 943
- $|$ (relation divise), 823
- \equiv (congru modulo n), 49, 1040
- $\not\equiv$ (non congru modulo n), 49
- $[a]_n$ (classe d'équivalence modulo n), 824
- $+_n$ (addition modulo n), 834
- \cdot_n (multiplication modulo n), 834
- $\left(\frac{a}{p}\right)$ (symbole de Legendre), 871
- ε (chaîne vide), 877, 943
- \sqsubset (relation préfixe), 877
- \sqsupset (relation suffixe), 877
- \triangleright (symbole de commentaire), 17
- \leq_P (réductibilité en temps polynomial), 951
- 2-CNF-SAT, 968
- 3-CNF, 965
- 3-CNF-SAT, 965
- abélien (groupe), 834
- absolument convergente
 - série, 1022
- absorption (lois d')
 - pour les ensembles, 1035

- abstrait (problème), 939
 - acceptation
 - état d', 886
 - par un algorithme, 944
 - par un automate fini, 886
 - accessibilité dans un graphe (\rightsquigarrow), 1044
 - ACM, 463
 - ACM-GÉNÉRIQUE, 547
 - ACM-KRUSKAL, 553
 - ACM-PRIM, 554
 - ACM-RÉDUIRE, 559
 - activités (problème de choix d'), 362–370
 - activités compatibles, 362
 - acyclique (graphe orienté)
 - et problème du chemin hamiltonien, 950
 - acyclique (graphe), 1044
 - additif (groupe), modulo n , 835
 - addition
 - d'entiers binaires, 19
 - de matrices, 704
 - des polynômes, 796
 - modulo n ($+_n$), 834
 - adjacents (sommets), 1044
 - admissible (arc), 660
 - admissible (réseau), 660, 661
 - ADN, 341, 342, 355
 - adressage ouvert (table de hachage à), 231–238
 - double hachage, 234, 235
 - sondage linéaire, 233
 - sondage quadratique, 233, 234, 244
 - affectation
 - multiple, 17
 - AFFICHAGE-PARENTHÉSAGE-OPTIMAL, 330
 - AFFICHER-POSTES, 322
 - agrégat (méthode de l'), 396–400
 - pour les compteurs binaires, 398, 399
 - pour les opérations de pile, 396–398
 - agrégat (méthode par)
 - pour le balayage de Graham, 922
 - pour les structures de données d'ensembles disjoints, 493, 494, 498
 - pour les tas de Fibonacci, 481
 - AKS (réseau de tri), 699
 - aléatoire
 - variable, 1069–1074
 - aléatoire (construction)
 - d'un ABR, 258–261
 - d'un arbre binaire de recherche, 263
 - aléatoire (machine à accès)
 - contre réseaux de comparaisons, 681
 - algorithme
 - temps d'exécution, 21
 - algorithme d'approximation
 - pour couplage maximum, 1014
 - pour couverture de sommets de poids minimal, 1004–1006
 - pour ordonnancement pour machines parallèles, 1015
 - pour problème de la satisfaisabilité MAX-CNF, 1007
 - pour problème MAX-CUT, 1007
 - pour satisfaisabilité MAX-3-CNF, 1003, 1004
 - randomisé, 1003
 - algorithme de Boruvka, 561
 - algorithme de couplage biparti de Hopcroft-Karp, 673
 - algorithme de Dijkstra
 - avec poids d'arc entiers, 583
 - implémenté via un tas min, 581
 - algorithme de Karmarkar, 752, 794
 - algorithme de Kruskal, 551–553
 - algorithme de l'ellipsoïde, 752
 - algorithme de préflot
 - heuristique du fossé pour, 669
 - algorithme de Prim, 554–556
 - dans algorithme d'approximation pour problème du voyageur de commerce, 993
 - sur des graphes peu denses, 558
 - algorithme du simplexe, 750, 765–779, 794
 - algorithme glouton
 - comparé à programmation dynamique, 333, 341, 365–367, 371, 372
 - pour ordonnancement de tâches, 392
 - algorithme randomisé, 89, 90
 - arrondi randomisé, 1017
 - et analyse probabiliste, 95, 96
 - et entrées moyennes, 24
 - pour le problème de l'embauche, 95, 96
 - pour permutation de tableau, 96–100
 - pour satisfaisabilité MAX-3-CNF, 1003, 1004
 - tri rapide, 154
- algorithmes randomisé,
 - algorithmique (géométrie), 901–932
 - allocation d'objets, 205–207

- ALLOUER-NŒUD, 433
 ALLOUER-OBJET, 207
alors, en pseudo code, 17
 alphabet, 886, 943
 alvéole, 216
 améliorant (chemin), 635, 636
 amortie (analyse), 395–419
 méthode comptable, 400–402
 méthode de l'agrégat, 396–400
 méthode du potentiel, 402–406
 pour algorithme préflot générique,
 pour l'algorithme de Knuth-Morris-Pratt,
 895
 pour la permutation miroir, 415
 pour le balayage de Graham, 922
 pour les arbres équilibrés pondérés, 416
 pour les piles sur un espace de stockage se-
 condaire, 442
 pour les structures de données d'en-
 sembles disjoints, 493, 494, 498
 pour les tables dynamiques, 406–414
 pour les tas de Fibonacci, 468–471, 476,
 477, 479–481
 pour structure de donnée d'ensembles dis-
 joints, 500–505
 pour une recherche dichotomique dyna-
 mique, 416
 analyse
 de la méthode diviser-pour-régner, 30, 31
 analyse (arbre d'), 965
 analyse amortie
 pour algorithme de Dijkstra, 581
 pour restructuration d'arbre rouge-noir,
 417
 pour structure de données d'ensembles
 disjoints, 492
 analyse des algorithmes, 19–25
 voir aussi analyse amortie, analyse proba-
 biliste, 19
 analyse par agrégat
 pour algorithme de Dijkstra, 581
 pour structure de données d'ensembles
 disjoints, 492
 analyse probabiliste, 89, 101–113
 d'algorithme d'approximation pour satis-
 faisabilité MAX-3-CNF, 1003, 1004
 de ballons et paniers, 104, 105
 de l'algorithme de Rabin-Karp, 884
 de suites, 106–110
 des collisions, 242
 du comptage probabiliste, 113
 du hachage parfait, 240–242
 du paradoxe des anniversaires, 101–104
 du partitionnement, 154
 du problème de l'embauche, 93, 94
 du tri rapide, 149–152, 154
 et algorithmes randomisés, 95, 96
 et entrées moyennes, 24
 pour sélection randomisée, 180–183
 ancêtre (commun)
 premier, 508
 ancêtre propre, 1050
 annulation
 lemme de l', 804
 vecteur d', 707
 annulation de flot, 629
 antisymétrie, 1039
 appartenance à un ensemble (\in), 1033
 appel
 de sous-routine, 19, 22
 par valeur, 18
 approximation (algorithme d'), 987–1017
 et rangement dans des boîtes, 1013
 par les moindres carrés, 737–740
 pour le problème de la couverture d'en-
 semble pondérée, 1014
 pour problème de la clique maximum,
 1013
 pour problème de la couverture d'en-
 semble, 997–1002
 pour problème de la couverture de som-
 mets, 989–992
 pour problème de la somme de sous-
 ensemble, 1007–1013
 pour problème du voyageur de commerce,
 992–997
 approximation (schéma d'), 988
 arbitrage, 596
 arborescence, 1050
 couvrante, 384, 385
 hauteur, 1051
 numérotée, 1052
 parcours d'une, 210
 représentation d'une, 208–211
 arborescence (parcours d'une), 210, 211
 arborescence binaire *voir* binaire (arbores-
 cence), 1051
 arborescences
 bissection d', 1054

- arbre, 1046–1053
 - arbre AA, 293
 - AVL, 289
 - binaire de recherche optimal, 347–354
 - binomial, 447, 448, 469
 - bouc-émissaire, 293
 - couverture de sommets optimale dans un, 992
 - couvrant, 545
 - d'analyse, 965
 - d'intervalles, 304–310
 - de décision, 160, 161
 - de fusion, 175, 423
 - de parcours en largeur, 518, 523
 - de parcours en profondeur, 525
 - de plus courts chemins, 568, 591–594
 - de rangs, 303
 - déployé, 293, 422
 - 2-3, 293, 444
 - 2-3-4, 430
 - diamètre d'un, 524
 - dynamique, 422
 - équilibré en hauteur, 289
 - équilibré en poids, 293
 - k -voisin, 293
 - parcours complet d'un, 995
 - parcours d'un, 249
 - récurtivité, 32
 - tas, 289
- arbre AA, 293
- arbre binaire de recherche, 247–265
 - voir aussi* rouge-noir (arbre), 247
 - arbre AA, 293
 - arbre AVL, 289
 - arbre k -voisin, 293
 - arbres bouc-émissaire, 293
 - arbres équilibrés en poids, 293
 - avec des clés égales, 262
 - clé maximale dans un, 251, 252
 - clé minimale dans un, 251, 252
 - construit aléatoirement, 258–261, 263
 - et treap, 289
 - insertion dans un, 254, 255
 - interrogation d'un, 250–254
 - optimal, 347–354
 - pour le tri, 257
 - prédécesseur dans un, 252, 253
 - recherche dans un, 250, 251
 - successeur dans un, 252, 253
 - suppression dans un, 255–257
- arbre binaire de recherche optimal, 349
- arbre bouc-émissaire, 293
- arbre couvrant
 - goulet d'étranglement, 560
 - vérification, 561
- arbre couvrant minimal
 - algorithme de Boruvka, 561
- arbre couvrant minimum
 - dans algorithme d'approximation pour problème du voyageur de commerce, 993
- arbre d'arité k complet
 - voir aussi* tas, 121
- arbre de fusion, 175, 423
- arbre de recherche équilibré
 - arbre AA, 293
 - arbre AVL, 289
 - arbre k -voisin, 293
 - arbres bouc-émissaire, 293
 - arbres équilibrés en poids, 293
 - treap, 289
- arbre de recherche exponentiel, 175, 423
- arbre de récursivité, 32
- arbre équilibré en poids, 293
- arbre k -voisin, 293
- arbre récursif
 - dans la démonstration du théorème général, 73, 74
 - et méthode de substitution, 67, 68
- arbre rouge-noir, 267–293
 - voir aussi* arbre
 - d'intervalles, rangs (arbre de), 267
 - clé maximale d'un, 270
 - clé minimale d'un, 270
 - énumération des clés d'un intervalle, 304
 - extension d'un, 302, 303
 - hauteur d'un, 270
 - insertion dans un, 273–281
 - jointure, 288
 - prédécesseur dans un, 270
 - propriétés d'un, 267–271
 - recherche dans, 270
 - relâché, 271
 - restructuration, 417
 - rotation dans un, 271–273
 - successeur dans un, 270
 - suppression dans un, 281–286
- ARBRE-INSÉRER, 255, 257, 273
- ARBRE-MAXIMUM, 252

- ARBRE-MINIMUM, 252
 ARBRE-PERSISTANT-INSÉRER, 287
 ARBRE-PRÉDÉCESSEUR, 253
 ARBRE-RECHERCHER, 250
 ARBRE-RECHERCHER-ITÉRATIF, 251
 ARBRE-SUCCESSEUR, 252
 ARBRE-SUPPRIMER, 256, 281
 Arbretas, 289
 arc
 admissible, 660
 arrière, 531
 avant, 531
 capacité d'un, 626
 classification dans un parcours en largeur, 542
 classification dans un parcours en profondeur, 531–533
 critique, 642
 de liaison, 523, 525, 531
 de poids négatif, 565, 566
 ensemble des, 1043
 poids d'un, 515
 pont, 542
 résiduel, 633
 saturé, 651
 transverse, 531
 arête
 minimale, 547
 sûre, 547
 argument d'une fonction, 1041
 arité
 d'une arborescence, 1052
 arithmétique
 congruence, 49
 série, 1023
 arithmétique avec infini, 570
 arrêt (problème de l'), 933
 arrière (arc), 531, 535
 arrière (substitution), 720
 arrondi, 1006
 randomisé, 1017
 articulation (sommets d'), 542
 assignation de vérité, 955, 962
 assignation satisfaisante, 955, 962
 associative (opération), 834
 associatives (lois)
 pour les ensembles, 1035
 asymptotique (notation)
 et algorithmes sur les graphes, 512
 asymptotiquement inférieure, 47
 asymptotiquement non négatif, 40
 asymptotiquement positive, 40
 asymptotiquement supérieure, 47
 AU-DESSOUS, 911
 AU-DESSUS, 910
 augmentation d'une clé dans un tas max, 132, 133
 AUGMENTER-CLÉ, 131
 AUGMENTER-CLÉ-TAS, 133
 automate
 fini, 885
 pour la recherche de chaîne de caractères, 886–891
 avant (arc), 531
 avant (substitution), 719, 720
 AVL-INSÉRER, 289
 B-arbre, 425–444
 clé minimum d'un, 437
 création, 433
 degré minimal d'un, 430
 2-3-4 (arbres), 430
 hauteur d'un, 430, 431
 nœud complet dans un, 430
 partage de nœud, 434, 435
 propriétés d'un, 429–431
 recherche, 432, 433
 suppression dans un, 439–442
 BALANCE, 289
 balayage, 908–915, 930
 balayage (droite de), 908
 état du, 910, 911
 BALAYAGE-GRAHAM, 919
 balise, 446
 ballons et paniers, 104, 105, 1085
 base, 341
 arbre à, 262
 d'une pile, 196
 tri par, 164–167
 Batcher (réseaux de fusion pair-impair de), 697
 Bayes (théorème de), 1067
 BELLMAN-FORD, 571
 Bellman-Ford (algorithme de), 571–575
 amélioration de Yen pour l', 595
 dans l'algorithme de Johnson, 620
 et fonctions objectifs, 588
 résolution de systèmes de contraintes de potentiel, 587

- Bernoulli (épreuve de), 1074
 BIASED-RANDOM, 90
 biconnexe (composante), 542
 bijective (fonction), 1042
 binaire
 algorithme du pgcd, 870
 fonction d'entropie, 1061
 relation, 1038
 binaire (arborescence), 1051
 complète, 1051
 parcours d'une, 210, 211
 binaire (arbre)
 nombre d'arbres binaires différents, 264
 binaire (compteur)
 analyse par la méthode comptable, 402
 analyse par la méthode de l'agrégat, 398, 399
 analyse par la méthode du potentiel, 404, 405
 de permutations miroir, 415
 et tas binomiaux, 461
 binomial
 coefficient, 1060
 développement, 1060
 binomial (arbre), 447, 448
 non trié, 469
 binomial (tas), 445–463
 clé minimale d'un, 451, 452
 création d'un, 451
 diminution d'une clé dans un, 458–460
 et compteur binaire et addition binaire, 461
 extraction de la clé minimale d'un, 458
 insertion dans un, 457, 458
 propriétés d'un, 449
 suppression dans un, 460, 461
 temps d'exécution des opérations sur un, 446
 union, 452–457
 binomiale (distribution), 1075–1078
 queues d'une, 1079–1085
 valeur maximum de la, 1078
 biparti (graphe), 1046
 couplage dans un, 485
 couplage maximum dans un, 644–648
 réseau de transport correspondant à un, 645
 bipartition (lemme de la), 805
 bissection d'une arborescence, 1054
 bitonique
 problème du voyageur de commerce, 354
 réseau de tri, 689–692
 séquence, 689
 tourné, 354
 bits (vecteur de), 217
 blanc
 sommet, 517, 525
 théorème du chemin, 530
 boîtes, 1013
 booléen
 connecteur, 962
 booléenne
 fonction, 1061
 formule, 950, 962, 968
 booléennes (multiplication des matrices)
 et fermeture transitive, 734
 borne
 asymptotiquement approchée, 40
 d'une sommation, 1025–1031
 garantie, 987
 inférieure asymptotique, 43
 polylogarithmique, 52
 polynomiale, 50
 pour distributions binomiales, 1077
 supérieure asymptotique, 42
 sur les coefficients binomiaux, 1060, 1061
 sur les queues d'une distribution binomiale, 1079–1085
 bouclage, de l'algorithme du simplexe, 777
 boucles, 1043
 B*-arbre, 430
 bureau de poste
 problème de l'emplacement du, 188

 câble, 682
 cache, 20
 CALCUL-FONCTION-PRÉFIXE, 895
 CALCUL-FONCTION-TRANSITION, 890
 capacité
 contrainte de, 626
 d'un arc, 626
 d'une coupure, 636
 résiduelle, 633, 635
 caractère (codage de), 376
 cardinal d'un ensemble ($| \cdot |$), 1036
 Carmichael (nombres de), 859, 865
 carré (élévations répétées au)
 et plus courts chemins pour tout couple de sommets, 607, 608
 pour calculer la puissance d'un nombre, 848

- carré d'un graphe orienté, 516
 carrée (matrice), 703
 carrée (racine)
 modulo un nombre premier, 871
 cartésien (produit), 1037
 cartésienne (somme), 803
 cas initial, 61
 cas le plus défavorable
 temps d'exécution dans le, 44
 cascade (coupe en), 479, 484
 Catalan (nombres de), 264, 325
 certificat
 d'un algorithme de vérification, 948
 dans un cryptosystème, 856
 chaînage, 243
 chaîne
 d'une enveloppe convexe, 923
 chaîne de caractères
 algorithme de Knuth-Morris-Pratt, 891–899
 algorithme de Rabin-Karp, 880–885
 au moyen d'automates finis, 885–891
 automate de, 886–891
 avec caractères joker, 880, 891
 basée sur les facteurs de répétition, 899
 dans une recherche de chaîne de caractères, 875
 recherche de, 875–900
 CHAÎNE-MATRICES-RÉCURSIF, 336
 chaînée (liste doublement), 199
 chaînée (liste),
 compacte, 207, 212
 insertion dans une, 200
 pour l'implémentation d'ensembles disjoints, 490–494
 recherche dans une, 200, 230
 suppression dans une, 200, 201
 changement
 de variable dans la méthode de substitution, 63, 64
 CHEMIN, 936, 944
 chemin, 1044
 augmentant, 635, 636, 673
 compression de, 495
 couverture, 670
 critique, 577
 dans une arborescence (longueur du), 1053
 hamiltonien, 950
 plus court, 520, 564
 plus long, 333, 934
 chemin de découverte, 494
 chemin élémentaire
 plus long, 333, 934
 chevauchement des sous-problèmes, 336–338
 chiffré (texte), 852
 chinois (théorème du reste), 843–846
 chirp (transformation), 811
 CHOIX-D'ACTIVITÉS-GLOUTON, 369
 CHOIX-D'ACTIVITÉS-RÉCURSIF, 367
 circuit
 dans un graphe, 1044
 de longueur négative *voir* négative (circuit de longueur), 565
 détection d'un, 536
 et plus courts chemins,
 pour transformée rapide de Fourier, 815, 816
 circuit (satisfaisabilité de), 954–960
 circuit combinatoire booléen, 955
 circuit hamiltonien, 934
 CIRCUIT-SAT, 956
 circulaire
 balayage, 916–924
 rotation, 899
 circulaire (liste chaînée)[*voir aussi* chaînée (liste)], 199
 classe
 d'équivalence, 1039
 de complexité, 945
 classe de complexité
 NP, 934
 NPC, 935
 P, 934
 classification des arcs
 dans un parcours en largeur, 542
 dans un parcours en profondeur, 531–533
 clé, 14, 117, 131, 192
 médiane, d'un nœud d'un B-arbre, 433
 publique, 851
 secrète, 851
 statique, 238
 clé médiane d'un nœud d'un B-arbre, 433
 CLIQUE, 970
 clique (problème de la), 969–972
 algorithme d'approximation pour le, 1013
 et problème de la couverture de sommets, 992
 CNF, 934

- co-NP, 949
- codage, 376, 377
 - binaire de caractère, 376
 - de Huffman, 376–383
 - de longueur fixe, 376
 - de longueur variable, 376
 - préfixe, 377
- coefficient
 - binomial, 1060, 1060, 1061
 - d'un polynôme, 50, 796
 - en forme relâchée, 758
- coefficients (représentation par), 797
 - et multiplication rapide, 801, 802
- cofacteur, 708
- cohérence des littéraux, 970
- colinéarité, 903, 907
- collision, 218
 - résolution par adressage ouvert, 231–238
 - résolution par chaînage, 219–222
- colonne
 - rang, 707
 - rang de, 707
 - vecteur, 702
- coloriage, 984
- combinaison ($\binom{n}{k}$), 1059, 1060
- combinatoire booléen (élément), 954
- comméragé, 419
- commun (diviseur), 824
 - plus grand *voir* plus grand commun diviseur, 824
- commune (sous-séquence)
 - plus longue, 341–347
- commutatives (lois)
 - pour les ensembles, 1034
- commutativité d'un opérateur, 834
- compacte (liste), 212
- comparables (segments de droite), 909
- comparaison (réseau de), 682–686
- comparateur, 682, 685, 694, 696
- comparatif (tri), 159
 - et arbres binaires de recherche, 250
 - et tas fusionnables, 477
- complément
 - d'un ensemble, 1035
 - d'un événement, 1064
 - d'un graphe, 973
 - d'un langage, 943
 - de Schur, 723, 736
- complet
 - graphe, 1046
 - nœud, 430
- complet (arbre binaire)
 - et un codage optimal, 377
- complet (parcours)
 - d'un arbre, 995
- complète
 - arborescence binaire, 1051, 1053
 - arborescence d'arité k , 1052
- complétude d'un langage, 961
- complexes (nombres)
 - multiplication des, 717
- complexes (racines de l'unité), 803
 - interpolation aux, 809, 810
- complexité (classe de), 945
 - co-NP, 949
 - NP, 948
 - NPC, 953
 - P, 941
- complexité (mesure de la), 945
- composante
 - biconnexe, 542
 - connexe, 1045
 - fortement connexe, 1045
- COMPOSANTES-CONNEXES, 489
- COMPOSANTES-FORTEMENT-CONNEXES, 538
- composé (nombre), 823
- comptable (méthode), 400–402
 - pour les compteurs binaires, 402
 - pour les opérations de pile, 401, 402
 - pour les tables dynamiques, 409
- comptage probabiliste, 113
- compteur *voir* binaire (compteur), 398
- concaténation
 - de langages, 943
 - des chaînes, 877
- conception
 - incrémentale, 25
 - méthode diviser-pour-régner, 26–31
- concret (problème), 940
- condition
 - aux limites, 60, 61
- conditionnelle
 - indépendance, 1069
 - probabilité, 1066, 1067
- congruence, modulo (\equiv), 49
- conjointe (fonction de densité), 1070
- conjonctive (forme normale), 965
- conjuguée (transposée), 734

- connecteur, 962
- connectivité, 644
- connexe (composante), 1045
 - identification grâce aux structures de données d'ensembles disjoints, 488–490
 - identifiée grâce au parcours en profondeur, 533
- connexe (graphe), 1045
- connexes (composantes fortement)
 - décomposition en, 536–542
- conservation du flot, 626
- consolidation d'une liste de racines dans un tas de Fibonacci, 472, 477
- CONSOLIDER, 475
- CONSTRUIRE-TAS-MAX, 126
- CONSTRUIRE-TAS-MAX', 135
- CONSTRUIRE-TAS-MIN, 128
- contour d'un polygone, 907
- contraction
 - d'un graphe non orienté selon un arc, 1046
- contraction d'un matroïde, 388
- contrainte
 - d'égalité, 588, 754, 755
 - d'inégalité, 754, 755
 - de positivité, 755
 - linéaire, 748
 - non-négativité, 753
 - stricte, 766
 - violation de, 766
- contrainte d'égalité, 588, 754
 - et contraintes d'inégalité, 755
 - stricte, 766
 - violation de, 766
- contrainte d'inégalité, 754
 - et contraintes d'égalité, 755
- contrainte de positivité, 753, 755
- contrainte linéaire, 748
- contrainte stricte, 766
- contraintes, 753
 - graphe de,
- convergente (série), 1022
- convexe
 - combinaison de points, 902
 - couche, 930
 - enveloppe, 915–925, 931
- convolution (\otimes), 798
 - théorème de, 810
- COPIE-INVERSION-BITS, 814
- côté d'un polygone, 907
- couches
 - convexes, 930
 - maximales, 930
- couleur d'un arbre rouge-noir, 267
- coup manqué, 883
- coupe
 - d'un graphe non orienté, 547
 - dans un réseau de transport, 636–638
 - dans un tas de Fibonacci, 479
 - en cascade, 479
 - ponds d'une, 1007
- COUPE-EN-CASCADE, 478
- COUPER, 478
- couplage
 - d'un graphe biparti pondéré, 485
 - et flot maximum, 644–648
 - maximum, 991, 1014
 - parfait, 648
- couplage biparti
 - algorithme de Hopcroft-Karp pour, 673
- couple, 1037
- courbe approchée, 737–740
- coût (amorti), 396, 400, 403
- couvert (sommet), 645
- couverture
 - de chemins, 670
 - de sommets, 972, 989, 1004–1006
 - par un sous-ensemble, 998
- couverture d'ensemble (problème de la), 997–1002
- couverture d'ensemble pondérée
 - problème de la, 1014
- couverture de sommet (problème de la)
 - NP-complétude du, 972, 973
- couverture de sommets (problème de la)
 - algorithme d'approximation pour, 989–992
 - et problème de la clique, 992
- couverture de sommets de poids minimal, 1004–1006
- couverture de sommets pondérée, 1004–1006
- COUVERTURE-ENSEMBLE-GLOUTON, 999
- COUVERTURE-SOMMET-APPROCHÉE, 990
- COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE, 1005
- COUVERTURE-SOMMETS, 972
- couvrant
 - arbre, 545
- couvrante
 - arborescence, 384, 385

- covertical, 910
 CPU (temps), 428
 CRÉER-ARBRE, 507
 CRÉER-B-ARBRE, 433
 CRÉER-ENSEMBLE, 488
 implémentation d'une forêt d'ensembles
 disjoints, 496
 implémentation en liste chaînée de, 491,
 493
 CRÉER-TAS, 445
 CRÉER-TAS-BINOMIAL, 451
 critique (arc), 642
 critique (chemin)
 dans un graphe orienté sans circuit, 577
 croisé (produit)
 \times , 903
 cryptosystème, 850–856
 cubique (spline), 742
 cycle, 1044
 de poids moyen minimal, 597
 et plus courts chemins,
 hamiltonien, 934, 946
 CYCLE-HAM, 947
 cyclique (groupe), 847
 cylindre, 426
- datation d'un sommet, 526
 date de fin de traitement
 et composantes fortement connexes, 539
 de poids minimum (arbre couvrant),
 de préflot (algorithmes), 649
 déchargement d'un sommet débordant de
 hauteur maximale, 669
 opérations élémentaires,
 pour trouver un couplage maximal dans un
 graphe bipartite, 658
 réétiqueter-vers-l'avant,
 de réduction, 937
 débordement
 d'un sommet, 649
 d'une pile, 196
 de file, 197, 198
 décalage dans une recherche de chaîne de ca-
 ractères, 875
 déchargement d'un sommet débordant, 661
 DÉCHARGER, 661
 décision
 arbre de, 160, 161
 par un algorithme, 944
 problèmes de, 940
 décision (arbre de)
 principe du zéro-un pour un, 689
 décomposition en valeurs singulières, 744
 DÉCOMPOSITION-LU, 725
 DÉCOMPOSITION-LUP, 728
 découvert (sommet), 517, 525
 DÉCRÉMENTER, 400
 DÉFILER, 197
 définie positive (matrice), 708
 dégénérescence, 777
 degré
 d'un nœud, 1051
 d'un polynôme, 49, 796
 d'un sommet, 1044
 de la racine d'un arbre binomial, 448
 maximal dans un tas de Fibonacci, 468,
 477, 482–484
 minimal d'un B-arbre, 430
 DeMorgan (lois de), 1035
 dénombrable (ensemble infini), 1036
 dénombrement, 1057–1063
 dans le tri par base, 166
 tri par, 162–164
 dense (graphe peu), 514
 dense (graphe), 514
 ε -dense, 621
 denses (non)
 distribution à enveloppes, 931
 DENSIFIER-LISTE, 208
 densité
 des nombres premiers, 857
 densité de probabilité
 fonction de, 1069
 dépendance, 707
 voir aussi indépendance, 707
 linéaire, 707
 dépilement
 opération de pile, 196
 DÉPILER, 196
 déployé (arbre), 293
 dérivation de série, 1024
 descendant, 1050
 destination (sommet de), 564
 det *voir* déterminant, 708
 déterminant, 708
 et multiplication des matrices, 734
 DÉTERMINER-RANG, 298
 déterministe, 95
 deux à deux

- ensembles disjoints, 1036
- entiers premiers, 826
- indépendance, 1066
- 2-3 (arbre), 293, 444
- 2-3-4 (arbre), 430
 - jointure, 443
 - scission, 443
- 2-3-4 (tas), 462
- deuxième arbre couvrant minimum, 558
- diagonale (matrice), 703
 - décomposition LUP d'une, 730
- diamètre d'un arbre, 524
- dichotomique (recherche)
 - avec insertion rapide, 416
- dictionnaire, 191
- différé (problème)
 - minimum, 506
 - premier ancêtre commun, 508
- différence d'ensembles ($-$), 1034
- différence symétrique, 673
- DIJKSTRA, 578
- Dijkstra (algorithme de), 577–583
 - avec pondération d'arc à valeurs entières, 582, 583
 - dans l'algorithme de Johnson, 620
 - implémenté à l'aide d'un tas de Fibonacci, 581
 - similitude avec algorithme de Prim, 554, 582
 - similitudes avec le parcours en largeur, 582
- DIMINUER-CLÉ, 131, 445
- DIMINUER-CLÉ-TAS-FIB, 478
- diminution d'une clé
 - dans un tas binomial, 458–460
 - dans un tas de Fibonacci, 478–481
 - dans un tas 2-3-4, 462
- direct (table à adressage), 216–218
- DIRECTION, 905
- discret (théorème du logarithme), 847
- discrète
 - distribution, 1064
 - variable aléatoire, 1069–1074
- disjoints (ensembles), 1036
 - analyse de la structure de données d', 500–505
 - cas particulier en temps linéaire, 509
 - dans algorithme de Kruskal, 553
 - et minimum différé, 506
 - et recherche de la profondeur, 507
 - implémentation en forêt des, 494–498
 - problème différé du premier ancêtre commun, 508
 - structure de données pour les, 487–509
- disjoints (forêt d'ensembles), 494–498
 - analyse d'une, 500–505
 - propriété du rang dans une, 500
- disjonctive (forme normale), 966
- disque, 914
 - voir aussi* secondaire (stockage), 425
- distance
 - d'édition, 355
 - de Manhattan, 188, 929
 - de plus court chemin, 520
 - euclidienne, 925
 - L_m , 929
- DISTANCE-MIN, 310
- distribution
 - à enveloppes non denses, 931
 - binomiale, 1075–1078
 - de probabilité, 1063–1065
 - de probabilité uniforme continue, 1065
 - des entrées, 89, 95
 - des nombres premiers, 857
 - géométrique, 1074, 1075
- distribution binomiale
 - et ballons et paniers, 105
- distribution géométrique
 - et ballons et paniers, 105
- distributivité (lois de)
 - pour les ensembles, 1035
- divergente (série), 1022
- divise (relation \mid), 823
- diviser pour régner (méthode)
 - pour la transformée rapide de Fourier, 806
 - pour le tri rapide, 139–157
 - pour l'algorithme de Strassen, 710–717
 - pour la conversion de binaire à décimal, 828
 - pour la multiplication, 816
 - pour trouver les deux points les plus rapprochés, 925–929
 - pour trouver une enveloppe convexe, 916
 - programmation dynamique, 315
- diviseur, 823
 - voir aussi* plus grand commun diviseur, 824
 - commun, 824
- division
 - essais de, 857

- méthode de la, 224, 225
- théorème de la, 823
- DNF (forme normale disjonctive), 966
- domaine de définition, 1040
- domaine de valeurs, 1040
- domaine des fréquences, 795
- domaine des temps, 795
- domine (relation), 930
- données (structure de),
 - arborescence, 208–211
 - arbre à base, 262
 - arbre binaire de recherche, 247–265
 - arbre d'intervalles, 304–310
 - arbre de rangs, 296–301
 - arbre déployé, 293, 422
 - arbre 2-3, 293, 444
 - arbre rouge-noir, 267–293
 - arbres dynamiques, 422
 - B-arbre, 425–444
 - dictionnaire, 191
 - extension d'une, 295–311
 - file, 195, 197
 - file à double entrée, 198
 - persistante, 287, 422
 - pile, 195–197
 - potentiel d'une, 403
 - pour les ensembles disjoints, 487–509
 - sur un espace de stockage secondaire, 426–429
 - table à adressage direct, 216–218
 - table de hachage, 218–223
 - tas binomial, 445–463
 - tas de Fibonacci, 465–485
 - tas 2-3-4, 462
 - tas relâché, 485
- dorsale, 289
- double hachage, 234, 235, 238
- droite
 - enfant, 1051
 - sous-arborescence, 1051
- DROITE, 122
- droite (segments de), 902
 - déterminer l'intersection de deux, 904–906
 - déterminer si des segments consécutifs tournent à gauche ou à droite, 904
 - déterminer si deux segments donnés se coupent, 908–915
 - trouver toutes les intersections dans un ensemble de, 915
- dualité, 779–785
 - faible, 780
- durée d'exécution, 392
- dynamique (arbre), 422
- dynamique (ensemble)
 - arborescence, 208–211
 - arbre à base, 262
 - arbre binaire de recherche, 247–265
 - arbre d'intervalles, 304–310
 - arbre de rangs, 296–301
 - arbre déployé, 293, 422
 - arbre 2-3, 293, 444
 - arbre rouge-noir, 267–293
 - B-arbre, 425–444
 - dictionnaire, 191
 - file, 197
 - file à double entrée, 198
 - file de priorité, 191
 - opérations de modification sur un, 192
 - persistant, 287
 - pile, 195–197
 - requête dans un, 192
 - structure de données pour les ensembles disjoints, 487–509
 - table à adressage direct, 216–218
 - table de hachage, 218–223
 - tas binomial, 445–463
 - tas de Fibonacci, 465–485
 - tas 2-3-4, 462
 - tas relâché, 485
 - vecteur de bits, 217
- dynamique (graphe)
 - algorithme de l'arbre couvrant minimum pour un, 557
 - fermeture transitive d'un, 621
- dynamique (programmation)
 - et fermeture transitive, 613–616
 - et plus courts chemins pour tout couple de sommets, 603–613
 - pour l'algorithme de Floyd-Warshall, 609–613
- dynamique (table), 406–414
 - analyse par la méthode comptable, 409
 - analyse par la méthode du potentiel, 409, 410, 412–414
 - facteur de remplissage d'une, 407
- dynamique (technique de la programmation), 315–360
 - chevauchement des sous-problèmes, 336–338

- comparaison avec les algorithmes glou-
tons, 373, 374
 - impression équilibrée, 355
 - pour algorithme de Viterbi, 358
 - pour la variante entière du problème du
sac-à-dos, 375
 - pour le choix d'activités, 370
 - pour multiplications matricielles enchaî-
nées,
 - pour plus longue sous-séquence commune,
341–347
 - pour problème bitonique du voyageur de
commerce, 354
 - recensement, 339–341
 - sous-structure optimale pour la, 331–336
- E [] (espérance), 1070
- e*, 50
- écart, 757
- écart-type, 1073
- écarts complémentaires, 792
- échange (propriété d'), 384
- échantillonnage, 147
- échec dans une épreuve de Bernoulli, 1074
- échelle (changement d')
- pour plus courts chemins à origine unique,
596
- échelonnement
- dans un flot maximum, 671
- ÉCRIRE-DISQUE, 428
- Edmonds-Karp (algorithme de), 641
- effet miroir, 806
- égalité des ensembles, 1033
- égalité entre fonctions, 1041
- égalité linéaire, 748
- élagage d'un tas de Fibonacci, 485
- ELAGUER-TAS-FIB, 485
- élément de l'ensemble (\in), 1033
- élémentaire
- chemin, 1044
 - événement, 1063
- élévations au carré
- et plus courts chemins pour tout couple de
sommets, 607, 608
- éloignés (problème des deux points les plus),
916
- sinon**, en pseudo code, 17
- EMBAUCHE-SECRÉTAIRE, 88
- EMBAUCHE-SECRÉTAIRE-RANDOMISÉ, 96
- emboîtée
- série, 1024
 - somme, 1024
- empilement
- opération de pile, 196
- EMPLER, 196
- encodage d'une instance de problème, 940–
943
- enfant, 1050, 1051
- liste dans un tas de Fibonacci, 467
- ENFILER, 197
- engendré (sous-graphe), 1045
- enregistrement (données), 117
- ensemble ($\{ \}$), 1033–1038
- indépendant, 983
- ensemble des valeurs, 1041
- ensemble disjoint (structure de données)
- pour l'ordonnement de tâches, 394
- ensemble dynamique, 191
- ensemble statique de clés, 238
- ENTASSER-MAX, 124
- ENTASSER-MIN, 126
- entière (variante)
- dans le problème du sac-à-dos, 373, 375
- entièrement polynomial (schéma d'approxi-
mation en temps), 988
- pour problème de la clique maximale,
1013
 - pour problème de la somme de sous-
ensemble, 1007–1013
- entières (flot à valeurs), 646
- entiers (\mathbf{Z}), 1034
- entiers naturels (\mathbf{N}), 1034
- entrant (degré), 1044
- entrée
- alphabet d', 886
 - câble d', 682
 - d'une porte logique, 954
 - de circuit combinatoire, 955
 - séquence d', 682
 - taille de l', 21
- entrées
- distribution des, 89, 95
- entropie (fonction), 1061
- épreuve de Bernoulli
- et ballons et paniers, 104, 105
 - et suites, 106–110
- épreuves (espace d'), 1063
- ε -dense (graphe), 621
- épuration d'une liste, 1009

- ÉPURER, 1010
- équation
de récurrence *voir* récurrence, 59
et notation asymptotique, 44, 45
- équilibré (arbre de recherche)
arbre déployé, 293, 422
arbre 2-3, 293, 444
arbre 2-3-4, 430
arbre rouge-noir, 267–293
arbres équilibrés pondérés, 416
B-arbre, 425–444
- équivalence (classe d'), 1039
modulo n ($[a]_n$), 824
- équivalence (relation d'), 1038, 1039
modulaire, 1040
- espérance
d'une distribution binomiale, 1076
d'une distribution géométrique, 1074
- ET (fonction) (\wedge), 614, 954
- ET (porte), 954
- et, en pseudo code, 18
- état d'un automate fini, 885
- étoile
polygone en forme d', 924
- EUCLIDE, 829
- Euclide (algorithme d'), 828–833, 871
- EUCLIDE-ÉTENDU, 831
- euclidienne
distance, 925
norme, 706
- Euler
fonction phi, 836
théorème d', 846, 865
- eulérien (circuit), 543
- évacuation (problème de l'), 669
- évaluation d'un polynôme, 36, 798, 802
en des points multiples, 818
et de ses dérivées, 818
- événement, 1063
échancier des points d', 910
- éventail, 955
- excédent de flot, 649
- exécution
d'une sous-routine, 22
- exécution (temps d')
d'un algorithme de graphe, 512
d'un réseau de comparaison, 683
- existence (problème d'), 584
- exploré (arc), 527
- exponentiation
modulaire, 848
- EXPONENTIATION-MODULAIRE, 849
- exponentielle, 50, 51
série, 1023
- extension
d'un ensemble, 385
d'une structure de données, 295–311
- EXTENSION-PLUS-COURTS-CHEMINS, 605
- extérieur
d'un polygone, 907
longueur du chemin, 1053
produit, 706
- externe (nœud), 1050
- extraction de la clé maximale
d'un tas max, 132
- extraction de la clé maximum
d'un tas d -aire, 135
- extraction de la clé minimale
dans un tableau de Young, 135
dans un tas binomial, 458
dans un tas de Fibonacci, 471–477
dans un tas 2-3-4, 462
- EXTRAIRE-MAX, 131, 132
- EXTRAIRE-MAX-TAS, 132
- EXTRAIRE-MIN, 131, 445
- EXTRAIRE-MIN-TAS-FIB, 472
- extrémité
d'un intervalle, 304
d'un segment de droite, 902
- facteur, 823
tournant, 808
- factorielle (!), 52
- factorisation, 865–870
unicité, 827
- Fermat (théorème de), 846
- fermeture
d'un langage, 943
propriété de groupe, 833
transitive, 613–616
- FERMETURE-TRANSITIVE, 614
- feuille, 1050
- FFT *voir* Transformée rapide de Fourier, 795
- FFT-ITÉRATIVE, 813
- FFT-RÉCURSIVE, 807
- Fibonacci (nombres de), 53, 54, 482
calcul des, 871
- Fibonacci (tas de), 465–485

- clé minimale dans un, 471
- création d'un, 469
- dans l'algorithme de Dijkstra, 581
- dans l'algorithme de Prim, 556
- degré maximal d'un, 468, 482–484
- diminution d'une clé dans un, 478–481
- élagage dans un, 485
- extraction de la clé minimale dans un, 471–477
- fonction potentiel pour les, 468
- insertion dans un, 469, 470
- modification d'une clé dans un, 485
- suppression dans un, 481, 484
- temps d'exécution des opérations sur un, 446
- FIFO (first-in, first-out), 195
 - voir aussi* file, 195
- fil, 955
- file, 195, 197
 - à double entrée, 198
 - dans l'algorithme pousser-réétiqueter, 669
 - dans le parcours en largeur, 518
- file de priorité, 131–134
 - voir aussi* arbre de recherche binaire, tas binomial, tas de Fibonacci, 131
 - avec extractions monotones, 136
 - file de priorité max, 131
 - file de priorité min, 131, 134
 - tas et, 131–134
- fil gauche, frère-droit, 211
 - représentation, 209
- final (fonction d'état), 886
- fini (automate), 885
 - pour la recherche de chaîne de caractères, 886–891
- fini (ensemble), 1036
- fini (groupe), 834
- finie (suite), 1041
- first-fit (algorithme), 1013
- first-in, first-out, 195
 - voir aussi* file, 195
- flot, 626–632
 - à coût minimal, 762, 763
 - à valeurs entières, 646
 - conservation de, 626
 - de transport, 627
 - excédent de, 649
 - maximum, 761
 - somme, 632
 - total, 764
 - valeur d'un, 627
- flot maximum & coupe minimum (théorème du), 638
- flot multi-ressources, 763, 764
 - coût minimal, 765
- FLOT-MAX-ET-ECHELONNEMENT, 672
- FLOYD-WARSHALL, 611
- Floyd-Warshall (algorithme de), 609–613, 615
- FLOYD-WARSHALL', 615
- fonction, 1040–1042
 - convexe, 1072
 - d'Ackermann, 509
 - génératrice, 81
 - itérée, 57
 - linéaire, 23, 747
 - objectif, 749, 753
 - quadratique, 23
- fonction de hachage
 - anti-collision, 855
 - ε -universelle, 230
- fonction potentiel
 - pour minorants, 419
- FORD-FULKERSON, 639
- Ford-Fulkerson (méthode de), 632–644
- forêt, 1046, 1047
 - d'ensembles disjoints, 494–498, 505
 - de parcours en profondeur, 525
- forme canonique, 748
- forme CNF, 965
- forme normale conjonctive d'ordre k , 934
- forme relâchée, 756–759
- forme standard, 748
 - unicité de la, 776
- formule (problème de la satisfaisabilité de), 962–964
- formule booléenne, 934
- formule satisfaisable, 934
- fortement connexe
 - composante, 1045
 - graphe, 1045
- fractionnaire (variante du problème du sac-à-dos), 375
- FUSION, 27
- fusion
 - à l'aide d'un réseau de comparaison, 692–694
 - de deux tableaux triés, 26
 - de k listes triées, 134

- minorant pour, 174
- fusion (réseaux de), 692–694
 - pair-impair, 697
- fusion (tri par)
 - implémentation en réseau de tri, 694–696
- fusionnable (tas), 445
 - voir aussi* binomial (tas), Fibonacci (tas de), 445
- dans l'algorithme de l'arbre couvrant minimum, 463
- tas 2-3-4, 462
- tas relâché, 485
- temps d'exécution des opérations sur un, 446
- FUSIONNER-LISTES, 1008
- FUSIONNER-TAS-BINOMIAUX, 452, 453
- FUSIONNEUR, 693
- Gabow (algorithme de)
 - pour plus courts chemins à origine unique, 596
- garantie (borne), 987
- garantie de performance, 1003
- GAUCHE, 122
- gauche
 - enfant, 1051
 - sous-arborescence, 1051
- Gauss (élimination de), 722
- générateur
 - de nombres aléatoires, 90
 - d'un sous-groupe, 838
 - de \mathbf{Z}_n^* , 847
- géométrique
 - distribution, 1074, 1075
 - série, 1023
- gestion de la mémoire, 121
- GLOUTON, 386
- glouton (algorithme), 361–394
 - algorithme de Dijkstra, 577–583
 - algorithme de Kruskal, 551–553
 - algorithme de Prim, 554–556
 - comparaison avec la programmation dynamique, 373, 374
 - et changeur de monnaie, 392
 - fondements théoriques, 383–389
 - pour arbre couvrant minimal, 546
 - pour l'ordonnancement de tâches, 389–392, 394
 - pour le choix d'activités, 362–370
 - pour le codage de Huffman, 376–383
 - pour le problème de la couverture d'ensemble pondérée, 1014
 - pour le problème du sac-à-dos fractionnaire, 374
 - pour problème de la couverture d'ensemble, 997–1002
 - pour trouver une couverture de sommet optimale dans un arbre, 992
 - propriété du choix glouton, 372
 - sous-structure optimale dans un, théorie, 370–376
- glouton (propriété du choix), 372
 - pour les codages de Huffman, 380, 381
- gloutonoïde, 394
- Goldberg (algorithme de) *voir* de préflots (algorithmes), 649
- goulot d'étranglement (problème du voyageur de commerce avec), 997
- graphe, 1043–1047
 - voir aussi* orienté (graphe), orienté (graphe sans circuit), non orienté (graphe), réseau, arbre, 1043
 - complément d'un, 973
 - d'intervalles, 370
 - dense, 514
 - des composantes, 538
 - ε -dense, 621
 - hamiltonien, 946
 - matrice d'incidence d'un, 393, 517
 - non hamiltonien, 946
 - parcours en largeur d'un, 517–525
 - parcours en profondeur d'un, 525–533
 - peu dense, 514
 - plus court chemin dans, 520
 - pondéré, 515
 - représentation par listes d'adjacences d'un, 514
 - représentation par matrice d'adjacences d'un, 515
 - simplement connexe, 533
 - tournée dans un, 978
- graphe acyclique orienté
 - et graphe des composantes, 538
- graphe des composantes, 538
- graphe dynamique
 - structures de données pour, 423
- graphe orienté
 - cycle hamiltonien d'un, 934
 - et plus longs chemins, 934

- tournée d'Euler d'un, 934
- GRAPHES-ISOMORPHES, 950
- graphique (matroïde), 384
- grappe
 - première, 233
 - secondaire, 234
- GREFFER, 507
- greffes successives (méthode par), 916
- grille, 669
- gris (sommet), 517, 525
- groupe, 833–839
 - cyclique, 847
- hachage, 215–245
 - adressage ouvert, 231–238
 - chaînage, 243
 - double, 238
 - k-universel, 244
 - parfait, 238–242
 - universel, 226–229
 - valeur de, 218
- hachage (fonction de), 218, 223–230
 - méthode de la division, 224, 225
 - méthode de la multiplication, 225, 226
- hachage (table de), 218–223
 - voir aussi* hachage, 218
 - dynamique, 414
- Hall (théorème de), 648
- hamiltonien
 - chemin, 950
 - cycle, 946
 - graphe, 946
 - problème du chemin, 982
 - problème du cycle, 947
- harmonique
 - nombre, 1023
 - série, 1023
- hauteur
 - d'un arbre binomial, 448
 - d'un arbre de décision, 161
 - d'un arbre rouge-noir, 270
 - d'un B-arbre, 430, 431
 - d'un nœud dans un arbre, 1051
 - d'un nœud dans un tas, 123, 129
 - d'un tas *d*-aire, 135
 - d'un tas de Fibonacci, 484
 - d'une arborescence, 1051
 - de tas, 123
 - exponentielle, 258
 - noire, 268
 - hauteur (fonction de)
 - dans les algorithmes pousser-réétiqueter, 650
 - hauteur exponentielle, 258
 - héréditaire (famille de sous-ensembles), 384
 - Hermitienne (matrice), 734
 - heure
 - d'achèvement, 1015
 - de début, 362
 - de disponibilité, 392
 - de fin, dans choix d'activités, 362
 - heuristique du fossé, 669
 - HOARE-PARTITION, 153
 - HOPCROFT-KARP, 674
 - Horner (règle de), 798
 - HUFFMAN, 379
 - Huffman (codage de), 376–383
 - hyperarc, 1046
 - hypergraphe, 1046
 - idempotence (lois d')
 - pour les ensembles, 1034
 - identité (matrice), 703
 - image, 1041
 - imbriquées (boîtes), 595
 - immédiate (problème de l'enveloppe convexe), 925
 - implicite (notation de sommation), 630
 - IMPRIMER-CHEMIN, 524
 - IMPRIMER-PLSC, 346
 - IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES, 603
 - IMPRIMER-SEGMENTS-SÉCANTS, 914
 - incidence, 1043, 1044
 - incidence (matrice d')
 - d'un graphe non orienté, 393
 - d'un graphe orienté, 393, 517
 - et contraintes de potentiel, 585
 - INCRÉMENTER, 398
 - incrémentielle (méthode de conception)
 - pour trouver une enveloppe convexe, 916
 - indépendance, 1066, 1069, 1070
 - des sous-problèmes en programmation dynamique, 335, 336
 - indépendante (famille de sous-ensembles), 384
 - indice d'un élément de \mathbf{Z}_n^* , 847
 - inégalité
 - de Boole, 1068
 - de Jensen, 1072

- linéaire, 748
- triangulaire
 - et arcs de poids négatifs, 997
 - pour plus courts chemins, 570, 589
- inférieure (matrice triangulaire), 704
- inférieures (bornes)
 - pour une enveloppe convexe, 924
- infini (ensemble), 1036
- infini, arithmétique avec, 570
- infinie
 - suite, 1041
- infixe (parcours), 250, 254
 - d'un arbre, 249
- initial (état), 886
- initiale (sous-matrice), 735
- INITIALISE-SIMPLEXE, 772, 786
- INITIALISER-PRÉFLOT, 652
- injective (fonction), 1042
- INSÉRER, 131, 445
- INSÉRER-ADRESSAGE-DIRECT, 217
- INSÉRER-B-ARBRE, 435
- INSÉRER-B-ARBRE-INCOMPLET, 436
- INSÉRER-HACHAGE, 232, 238
- INSÉRER-HACHAGE-CHAÎNÉE, 219
- INSÉRER-INTERVALLE, 305
- INSÉRER-LISTE, 200
- INSÉRER-LISTE', 202
- INSÉRER-TABLE, 408
- INSÉRER-TAS-FIB, 470
- INSÉRER-TAS-MAX, 134, 135
 - construction d'un tas avec, 135
- INSÉRER-TAS-MIN, 134
- Insert, 406
- INSERTION, 192
- insertion
 - dans tableau de Young, 135
 - dans tas, 133, 134
 - dans un arbre binaire de recherche, 254, 255
 - dans un arbre d'intervalles, 306
 - dans un arbre de rangs, 299, 300
 - dans un arbre rouge-noir, 273–281
 - dans un B-arbre, 435–437
 - dans un état de droite de balayage, 910
 - dans un tas binomial, 457, 458
 - dans un tas d -aire, 135
 - dans un tas de Fibonacci, 469, 470
 - dans un tas 2-3-4, 462
 - dans une file, 197
 - dans une pile, 195
 - dans une table à adressage direct, 217
 - dans une table à adressage ouvert, 231, 232
 - dans une table de hachage, 219
 - dans une table dynamique, 408
- insertion (tri par)
 - dans le tri rapide, 153
 - implémentation en réseau de tri du, 685
- instabilité numérique, 718
- instance
 - de problème abstrait, 939
- instruction
 - addition, 19
 - de stockage, 19
 - division, 19
 - exponentiation, 20
 - partie entière supérieure, 19
 - arithmétique, 19
 - de branchement, 19
 - de branchement conditionnel, 19
 - de branchement inconditionnel, 19
 - de contrôle, 19
 - de copie, 19
 - de décalage, 20
 - de lecture, 19
 - de sortie de sous-routine, 19
 - de transfert de données, 19
 - modèle RAM, 19
 - modulo, 19
 - multiplication, 19
 - partie entière, 19
 - soustraction, 19
- intégralité (théorème de l'), 647
- intégration de série, 1024
- intérieur
 - longueur du chemin, 1053
- intérieur d'un polygone, 907
- intermédiaire (sommet), 610
- interne (nœud), 1050
- interpolation par un polynôme, 798
 - aux racines complexes de l'unité, 809, 810
- interpolation par une spline cubique, 742
- intersection
 - d'ensembles (\cap), 1034
 - dans un ensemble de segments de droite, 915
 - de cordes, 301
 - de deux disques, 914
 - de deux polygones simples, 914
 - de deux segments de droite, 904–906

- de langages, 943
- recherche dans un ensemble de segments de droite d'une, 908–915
- INTERSECTION-DEUX-SEGMENTS-QUELCONQUES, 911
- INTERSECTION-SEGMENTS, 905
- intervalle, 304, 305
 - fermé, 304
 - ouvert, 304
 - semi-ouvert, 304
 - tri flou d'un, 156
- intervalles
 - recouvrement, 304
 - trichotomie, 304
- intervalles (graphe d')
 - problème du coloriage d'un, 370
- intraitabilité, 933
- invalide (décalage), 875
- invariant de boucle, 15
 - conservation, 16
 - et boucles **pour**, 16
 - finalisation, 16
 - historique, 37
 - initialisation, 16
 - pour algorithme d'arbre couvant minimal, 547
 - pour algorithme de Dijkstra, 579
 - pour algorithme de préflot générique, 655
 - pour algorithme de Prim, 555, 556
 - pour algorithme de Rabin-Karp, 883
 - pour algorithme du simplexe, 773
 - pour algorithme réétiqueter-vers-l'avant, 665
 - pour augmentation d'une clé dans un tas, 134
 - pour consolidation de la liste de racines lors de l'extraction du nœud minimal d'un tas de Fibonacci, 475
 - pour construction d'un tas, 126
 - pour détermination du rang d'un élément dans un arbre de rangs, 298
 - pour fusion, 27–29
 - pour insertion dans arbre rouge-noir, 276
 - pour parcours en largeur d'un graphe, 519
 - pour partitionnement, 140, 141
 - pour permutation aléatoire de tableau, 99
 - pour recherche dans un arbre d'intervalles, 308
 - pour règle de Horner, 36
 - pour tri par insertion, 15–17
 - pour tri par tas, 130
 - pour union de tas binomiaux, 461
- inverse
 - d'une matrice, 706, 709
 - d'une matrice à partir d'une décomposition LUP, 731
 - multiplicatif modulo n , 842
- inverse d'une fonction bijective, 1042
- inverseur, 954
- inversible (matrice), 706
- inversion dans une séquence, 36
- inversion dans une suite, 94
- isolé (sommet), 1044
- isomorphes (graphes), 1045
- itération fonctionnelle, 53
- itérative (méthode),
- Jarvis (parcours de), 922–924
- JOHNSON, 620
- Johnson (algorithme de), 616–621
- jointure
 - d'arbres 2-3-4, 443
 - d'arbres rouge-noir, 288
- joker (caractère), 880
- Josephus (permutation de), 311
- k-chaîne, 1058
- k-CNF, 934
- k-combinaison, 1059
- k-permutation, 1059
- k-sous-chaîne/k-sous-chaîne, 1058
- k-sous-ensemble, 1036
- k-trié, 174
- k-universel (hachage), 244
- Karp (algorithme de)
 - pour cycle de poids moyen minimal, 597
- k ième (puissance), 827
- Kleene
 - opération étoile (*), 943
- KMP, 894
- Knuth-Morris-Pratt (algorithme de), 891–899
- Kraft (inégalité de), 1053
- Kruskal (algorithme de)
 - avec poids entiers, 557
- Lagrange
 - formule de, 799
 - théorème de, 837
- Lamé (théorème de), 830

- langage, 943
 complétude d'un, 961
 démonstration de la NP-complétude d'un, 961, 962
 vérification d'un, 948
 largeur (arbre de parcours en), 518, 523
 largeur (parcours en)
 et plus courts chemins, 520–523, 564
 similitudes avec l'algorithme de Dijkstra, 582
 last-in, first-out, 195
 voir aussi pile, 195
 Legendre (symbole de) ($\frac{a}{p}$), 871
 lemme de Farkas, 793
 lexicographique (tri), 262
 lg (logarithme de base 2), 51
 lg lg (composition), 51
 lg^* (fonction logarithme itéré), 53
 lg^k (exponentiation), 51
 liaison
 arc de, 523, 525, 531
 matrice de, 602
 liaison (sous-graphe de)
 pour les plus courts chemins parmi tout couple de sommets, 602
 libération d'objets, 205–207
 LIBÉRER-OBJET, 206
 libre (liste), 206
 lien
 entre deux arbres binomiaux, 447
 entre les racines d'un tas de Fibonacci, 472
 LIEN-BINOMIAL, 452
 LIER, 496
 LIFO (last-in, first-out), 195
 voir aussi pile, 195
 ligne
 rang, 707
 vecteur, 702
 linéaire
 dépendance, 707
 indépendance, 707
 ordre, 1040
 sondage, 233
 linéaire (programmation)
 et plus courts chemins à origine unique, 583–589
 linéaires (équations)
 résolution de système tridiagonal de, 741
 résolution de systèmes d', 717–730
 linéarité
 de l'espérance, 1071
 des sommes, 1022
 LIRE-DISQUE, 428
 liste *voir* chaînée (liste), 199
 liste à saut, 293
 littéral, 965
 L_m (distance), 929
 ln (logarithme népérien), 51
 logarithme,
 discret, 847
 itéré (lg^*), 53
 logique (porte), 954
 longueur
 d'un chemin, 563, 1044
 d'une chaîne, 1058
 d'une dorsale, 289
 d'une suite, 1041
 poids d'un, 563
 longueur (fonction), 386
 LONGUEUR-PLSC, 345, 347
 LONGUEUR-PLUS-LONG-CHEMIN, 945
 LU (décomposition), 722–725
 LUP (décomposition), 718
 calcul d'une, 725–728
 d'une matrice de permutation, 730
 d'une matrice diagonale, 730
 et multiplication des matrices, 734
 pour l'inversion des matrices, 731
 utilisation de la, 718–722
 machine à accès aléatoire, 19, 20
 Manhattan (distance de), 188, 929
 Markov (inégalité de), 1073
 marqué (nœud), 467, 479, 481
 matrice, 702–710
 voir aussi matrices (inversion des), matrices (multiplication des), 702
 d'adjacences, 515
 d'incidence, 393, 517
 de liaison, 602
 de Toeplitz, 817
 Hermitienne, 734
 pseudo-inverse d'une, 739
 symétrique définie positive, 735–737
 transposée conjuguée d'une, 734
 transposée d'une, 515, 702
 matrice symétrique, 709
 matrices (inversion des), 731–734
 matrices (multiplication des),

- algorithme de Strassen, 710–717
- booléennes, 734
- et calcul du déterminant, 734
- et décomposition LUP, 734
- et inversion des matrices, 731–734
- et plus courts chemins pour tout couple de sommets, 603–609
- méthode de Pan, 717
- matriciel (matroïde), 384
- MATRIX-CHAIN-MULTIPLY, 330
- matroïde, 383–389, 393
- MAUVAISE-INSTANCE-COUVERTURE-ENSEMBLE, 1002
- maximal
 - élément, d'un ensemble partiellement ordonné, 1039
 - point, 930
 - sous-ensemble, dans un matroïde, 385
- maximal (degré)
 - dans un tas de Fibonacci, 468, 477, 482–484
- maximales (couches), 930
- MAXIMUM, 131, 132, 192
- maximum, 177
 - d'un arbre rouge-noir, 270
 - dans tas, 132
 - dans un arbre binaire de recherche, 251, 252
 - dans un arbre de rangs, 303
 - de la distribution binomiale, 1078
 - recherche du, 178, 179
- maximum (couplage)
 - dans un graphe biparti, 644–648, 658
- maximum (flot),
 - algorithme de préflots, 649
 - algorithme d'échelonnement, 671
 - algorithme réétiqueter-vers-l'avant, 659–669
 - avec capacités négatives, 672
 - et couplage maximum dans un graphe biparti, 644–648
 - méthode de Ford-Fulkerson, 632–644
 - mise à jour du, 671
- MAXIMUM-EN-LIGNE, 111
- MAXIMUM-TAS, 132
- médian, 177–189
 - de liste triée, 187
 - parmi 3 (méthode du), 156
 - pondéré, 188
- MÊME-COMPOSANTE, 489
- mémoire
 - hiérarchisée, 20
 - virtuelle, 20
- MÉMORISATION-CHAÎNE-MATRICES, 339
- méthode
 - de Akra-Bazzi pour la résolution des récurrences, 84, 85
 - de substitution, 60–64
 - méthode de point intérieur, 752
 - méthode de programmation dynamique comparé à algorithmes gloutons, 333, 341, 371, 372
 - pour arbres binaires de recherche optimaux, 347–354
 - pour distance d'édition, 355
 - pour ordonnancement, 359
 - pour ordonnancement de chaînes de montage, 316–323
 - reconstruction d'une solution optimale dans la, 338, 339
 - méthode de substitution
 - et arbre récursif, 67, 68
 - méthode diviser-pour-régner
 - et arbre récursif, 64
 - méthode du potentiel
 - pour restructuration d'arbre rouge-noir, 417
- MÉTHODE-FORD-FULKERSON, 633
- MILLER-RABIN, 861
- Miller-Rabin (test de primalité), 859–865
- mineur d'une matrice, 707
- minimal (cycle de poids moyen), 597
- minimal (degré)
 - d'un B-arbre, 430
- minimal (nœud)
 - d'un tas de Fibonacci, 468
- minimale (arborescence couvrante)
 - et matroïdes, 386
- minimale (arête), 547
- minimale (coupe)
 - d'un réseau de flot, 636
- MINIMUM, 131, 178, 192, 445
- minimum, 177
 - d'un arbre rouge-noir, 270
 - d'un B-arbre, 437
 - dans un arbre binaire de recherche, 251, 252
 - dans un arbre de rangs, 303
 - dans un tas binomial, 451, 452

- dans un tas de Fibonacci, 471
 - différé, 506
 - recherche du, 178, 179
- minimum (arbre couvrant)
 - algorithme de Kruskal,
 - algorithme de Prim, 554–556
 - algorithme générique, 546–551
 - construit à l'aide de tas fusionnables, 463
 - dans un graphe dynamique, 557
 - deuxième, 558
- minimum (clé)
 - dans un tas 2-3-4, 462
- minimum (couverture)
 - de chemins, 670
- MINIMUM-DIFFÉRÉ, 507
- MINIMUM-TAS-BINOMIAL, 451, 461
- minorant
 - et fonction potentiel, 419
 - pour couverture de sommets de poids minimal, 1006
 - pour fusion, 174
 - pour le tri, 159–162
 - pour recherche du médian, 189
 - pour taille de couverture de sommets optimale, 991
 - pour tri par moyenne, 174
- miroir (permutations), 415
 - compteur binaire de, 415
- MIROIR-INCRÉMENTER, 415
- modification d'une clé
 - dans un tas de Fibonacci, 485
- modulaire
 - exponentiation, 848
- modulo, 49
- moindres carrés (approximation par les), 737–740
- monnaie (problème du changeur de), 392
- monotonie
 - croissante, 49
 - décroissante, 49
 - stricte, 49
- motif
 - dans un graphe, 974
- moyen (poids)
 - d'un cycle, 597
- moyenne
 - valeur, 1070–1072
- moyenne *voir* espérance, 1070
- multi ensemble, 1033
- MULTIDÉP, 397
- MULTIEMP, 400
- multigraphe, 1046
 - conversion en son graphe non orienté équivalent, 516
- multiple, 823
 - d'un élément modulaire n ,
 - d'un élément modulo n ,
- multiples (sources et puits), 629
- multiplicatif (groupe), modulo n , 835
- multiplicatif (inverse)
 - modulo n , 842
- multiplication
 - de matrices, 705
 - de nombres complexes, 717
 - des polynômes, 796
 - méthode de la, 225, 226
 - méthode diviser-pour-régner pour la, 816
 - modulo n (\cdot_n), 834
- MULTIPLIER-MATRICES, 324, 606
- Multipush, 400
- mutuellement exclusifs
 - événements, 1063
- mutuellement indépendants
 - événements, 1066
- N** (ensemble des entiers naturels), 1034
- n -aire (tas), 621
 - dans les algorithmes de plus court chemin, 621
- n -uplet, 1036, 1037
- nœud
 - voir aussi* sommet, 1050
 - d'une spline, 742
- naïf (algorithme)
 - pour la recherche de chaîne de caractères, 878–880
- naturelle (spline cubique), 742
- négatif (arc de poids)
 - dans l'algorithme de Dijkstra, 582
 - et plus courts chemins, 565, 566
- négatif (circuit de poids)
 - détection par l'algorithme de Floyd-Warshall, 616
 - et contraintes de potentiel, 586
 - recherche d'un, 609
- négatif (cycle de poids)
 - et relâchement, 594
- négatif (détection d'un circuit de poids)
 - avec un algorithmes des plus courts chemins pour tout couples de sommets, 609

- négative (circuit de longueur)
 et plus court chemin, 565
 recherche d'un, 575
 net (flot)
 à travers une coupe, 636
 neutre (élément), 833
 NIL, 18
 niveau d'une fonction, 498
 noir (sommet), 517, 525
 noire (hauteur), 268
 nombre d'or (ϕ), 54
 nombres réels (\mathbf{R}), 1034
 NON (fonction) (\neg), 954
 non dénombrable (ensemble), 1036
 non déterministe (temps polynomial), 948
 voir aussi NP, 948
 non hamiltonien (graphe), 946
 non inversible (matrice), 706
 non orienté (graphe), 1043
 clique dans un, 969
 coloration d'un, 984, 1054
 composante biconnexe d'un, 542
 conversion en, à partir d'un multigraphe,
 516
 couplage dans un, 645
 couverture de sommets d'un, 972, 989
 d-régulier, 648
 ensemble indépendant d'un, 983
 grille, 669
 point d'articulation d'un, 542
 pont d'un, 542
 non orientée (version)
 d'un graphe orienté, 1045
 non recouvrable (chaîne de caractères), 891
 non saturant (poussage), 657
 non singulière (matrice), 706
 non trié (arbre binomial), 469
 non triée (liste chaînée)[*voir aussi* chaînée
 (liste)], 199
 non triviale (puissance), 827
 non triviale (racine carrée de 1)
 modulo n , 848
 non truquée (pièce), 1065
 normale (équation), 739
 norme d'un vecteur, 706
 NOT (porte), 954
 notation Θ
 symétrie, 47
 notation asymptotique, 40–48, 56
 et linéarité des sommes, 1022
 réflexivité, 47
 symétrie transposée, 47
 transitivité, 47
 notation grand O, 41–43
 notation grand oméga, 43, 44
 notation O , 41–43
 notation o , 46
 notation O' , 56
 notation \tilde{O} , 56
 notation Omega, 41
 notation petit o, 46
 notation petit oméga, 46
 noyau d'un polygone, 924
 NP (classe de complexité), 934, 948, 950
 NP-complet, 935, 953
 NP-complétude, 933–986
 d'un langage (démonstration de la), 961,
 962
 de l'ordonnancement avec profits et dates
 d'échéance, 985
 du problème de la coloration de graphe,
 984
 du problème de l'ensemble stable maxi-
 mum, 983
 du problème de la clique, 969–972
 du problème de la couverture d'ensemble,
 1002
 du problème de la couverture de sommets,
 972, 973
 du problème de la programmation entière
 0-1, 982
 du problème de la programmation linéaire
 entière, 982
 du problème de la satisfaisabilité 3-CNF,
 965–968
 du problème de la satisfaisabilité de cir-
 cuit, 954–960
 du problème de la satisfaisabilité de for-
 mule, 962–964
 du problème de la semi satisfaisabilité 3-
 CNF, 983
 du problème de savoir si une formule boo-
 lénne est une tautologie, 968
 du problème du chemin hamiltonien, 982
 du problème du partitionnement d'en-
 semble, 982
 du problème du plus long cycle élémen-
 taire, 983

- du problème du sous-graphe isomorphe, 982
- du problème du voyageur de commerce, 978, 979
- NP-difficile, 953
- NPC (classe de complexité), 935, 953
- numérique
 - instabilité, 718
 - signature, 852
- objectif, 979
- objectif (fonction), 583, 588
- objet, 18
 - allocation et libération, 205–207
 - attribut, 18
 - champ, 18
 - implémentation par tableau, 203–208
 - passé en paramètre, 18
- occurrence d'une chaîne de caractères, 875
- ombre d'un point, 924
- opérateur court-circuitant, 18
- opposé
 - d'une matrice, 705
- optimal (sous-ensemble)
 - d'un matroïde, 385
- optimale (couverture de sommets), 989
- optimale (sous-structure)
 - d'un plus court chemin, 610
 - d'une plus longue sous-séquence commune, 342, 343
 - dans les algorithmes gloutons, de matroïdes pondérés, 388
 - des codages de Huffman, 381, 382
 - des plus courts chemins, 604
 - du problème du sac-à-dos, 373, 374
 - en programmation dynamique, 331–336
 - pour multiplications matricielles enchaînées, 325, 326
- optimisation (problèmes d'), 315, 940
 - algorithmes d'approximation pour, 987–1017
- ordonnancement, 359, 392, 985, 1015
 - de chaînes de montage, 316–323
- ordonnée (arborescence), 1051
- ordre
 - d'un groupe, 838
 - linéaire, 1040
 - partiel, 1039
 - total, 1040
- ordre de grandeur, 24
- ORDRE-CHAÎNE-MATRICES, 327
- orienté (graphe sans circuit)
 - algorithme du plus court chemin à origine unique pour un, 575–577
 - et problème du chemin hamiltonien, 950
- orienté (graphe sans circuit) et arcs arrière, 535
- tri topologique d'un, 1043
- orienté (graphe), 1043
 - carré d'un, 516
 - circuit eulérien d'un, 543
 - couverture de chemins minimum, 670
 - diagramme PERT, 576, 577
 - fermeture transitive d'un, 613
 - graphe de potentiel, 585
 - plus court chemin dans un, 564
 - plus courts chemins à origine unique dans un, 563–599
 - plus courts chemins pour tout couple de sommets dans un, 601–623
 - semi-connexe, 542
 - simplement connexe, 533
 - transposée d'un, 516
- orienté (segment), 902–904
- orientée (version)
 - d'un graphe non orienté, 1045
- origine, 517, 564, 902
- orthonormal, 744
- OU (fonction) (\vee), 614, 954
- OU (porte), 954
- OU, en pseudo code, 18
- P (classe de complexité), 934, 941, 945, 946
- PAC, 508
- page sur un disque, 427, 442
- pagination, 20
- pair-impair
 - réseau de fusion, 697
 - réseau de tri, 697
- paire d'entiers acceptable, 863
- Pan (méthode), pour la multiplication des matrices, 717
- papillon (opération), 811
- paquet cadeau, 922
- paquets, 168
- paramètre, 18
 - coût du passage de, 81
- parcours
 - d'un arbre, 298
 - d'une arborescence, 210, 211

- en largeur, 517–525
- infixe, 249
- postfixe, 249
- préfixe, 249
- parcours en profondeur, 525–533
 - lors d'un tri topologique, 534–536
 - lors de la recherche de composantes fortement connexes, 536–542
 - pour trouver les points d'articulation, les ponts, et les composantes biconnexes, 542
- parcours infixe d'un arbre, 298
- PARCOURS-INFIXE, 249
- PARENT, 122
- parent, 1050
 - dans un arbre de parcours en largeur, 518
- parenthésage d'un produit de matrices, 323
- parenthésé (entièrement), 323
- parenthèses (théorème des), 528
- partage
 - de nœud de B-arbre, 434, 435
- PARTAGER-ENFANT-B-ARBRE, 434
- fonction partie entière ($\lfloor \]$), 49
 - dans théorème général,
- fonction partie entière supérieure ($\lceil \]$), 49
- partiel
 - ordre, 1039
- parties (ensemble des), 1036
- PARTITION, 140
- partition d'un ensemble, 1036, 1039
- PARTITION-RANDOMISE, 148
- partitionnement
 - algorithme de, 140–142
 - algorithme randomisé de, 148
 - autour de l'élément médian parmi trois (algorithme de), 153
- partitionnement d'un ensemble
 - problème du, 982
- Pascal (triangle de), 1062
- passé (méthode une), 509
- PATH-HAM, 950
- performances
 - asymptotiques, 39
- permutation, 1042
 - de Josephus, 311
 - aléatoire, 96–100
 - aléatoire uniforme, 89, 97
 - d'un ensemble, 1059
 - directe, 98
 - miroir, 415
 - permutation (matrice de), 704, 709
 - décomposition LUP d'une, 730
 - permutation (réseau de), 698
 - permutation aléatoire, 96–100
 - uniforme, 89, 97
 - permutation aléatoire uniforme, 89, 97
 - PERMUTE-AVEC-TOUT, 100
 - PERMUTE-PAR-CYCLE, 100
 - PERMUTE-PAR-TRI, 97
 - PERMUTE-SANS-IDENTITÉ, 100
 - persistante (structure de données), 287, 422
 - PERT (diagramme), 576, 577
 - PEUT-ÊTRE-ACM-A, 560
 - PEUT-ÊTRE-ACM-B, 561
 - PEUT-ÊTRE-ACM-C, 561
 - pgcd, 824–826
 - voir aussi* plus grand commun diviseur, 824
 - phi (fonction), 836
 - pile, 195–197
 - d'exécution de la procédure, 155
 - dans le balayage de Graham, 918
 - opérations analysées avec la méthode comptable, 401
 - opérations analysées avec la méthode de l'agrégat, 396–398
 - opérations analysées avec la méthode du potentiel, 404
 - sur un espace de stockage secondaire, 442
 - vide, 196
 - PILE-VIDE, 196
 - PIVOT, 770
 - pivot, 724
 - dans le tri rapide, 140
 - en programmation linéaire, 768, 770, 771, 778
 - PL, 518
 - plein (rang), 707
 - PLSC (plus longue sous-séquence commune), 341–347
 - plus court chemin, 563–599, 601–623
 - à destination unique, 564
 - à origine unique, 563–599
 - à paire unique, 333
 - algorithme de Bellman-Ford, 571–575
 - algorithme de changement d'échelle de Gabow, 596
 - algorithme de Dijkstra, 577–583
 - algorithme de Floyd-Warshall, 609–613

- algorithme de Johnson, 616–621
- arbre de, 568, 591–594
- avec arcs de poids négatif, 565, 566
- avec chemins bitoniques, 599
- dans graphe non pondéré, 333
- dans un graphe ε -dense, 621
- dans un graphe non pondéré, 520
- dans un graphe orienté acyclique, 575–577
- dans un graphe pondéré, 564
- estimation de, 568
- et circuit de longueur négative, 565
- et contraintes de potentiel, 583–589
- et parcours en largeur, 520–523, 564
- et produit de matrices, 603–609
- et relâchement, 568–570
- inégalité triangulaire de, 570
- par élévations au carré, 607, 608
- pour tout couple de sommets, 564
- pour un couple de sommets unique, 564
- propriété aucun-chemin de, 570, 590
- propriété de convergence de, 570, 590, 591
- propriété de majorant de, 570, 590
- propriété de relâchement de chemin, 591
- propriété de relâchement de chemin de, 570
- propriété de sous-graphe prédécesseur de, 570, 593, 594
- sous-structure optimale d'un, 565
- variantes au problème du, 564
- plus court chemin à couple unique
 - en tant que programme linéaire, 761
- plus court chemin à origine unique
 - avec chemins bitoniques, 599
 - et plus longs chemins, 934
- plus court chemin non pondéré, 333
- plus court chemins
 - inégalité triangulaire de, 589
- plus courts chemins
 - en tant que programme linéaire, 761
 - et plus longs chemins, 934
- plus courts chemins pour tout couple de sommets, 601–623
 - algorithme de Floyd-Warshall, 609–613
 - algorithme de Johnson, 616–621
 - dans un graphe ε -dense, 621
 - et produit de matrices, 603–609
 - par élévations au carré, 607, 608
- plus grand commun diviseur (pgcd), 824–826
 - avec plus de deux arguments, 833
 - calculé avec l'algorithme d'Euclide, 828–833
 - calculé par l'algorithme du pgcd binaire, 870
 - théorème de récursivité pour le, 828
- plus long chemin élémentaire
 - dans graphe non pondéré, 333
- plus long chemin élémentaire non pondéré, 333
- plus long chemin simple, 934
- plus long cycle simple (problème du), 983
- plus longue sous-séquence commune, 359
- plus petit commun multiple, 833
- PLUS-COURT-CHEMIN, 936
- PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ, 608, 609
- PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI, 606
- PLUS-COURTS-CHEMINS-GSS, 575
- PLUS-LONG-CHEMIN, 945
- PLUS-RAPIDE-CHEMIN, 321
- poids
 - d'un arc, 515
 - d'une coupure, 1007
 - moyen, 597
- poignée de main (lemme de la), 1047
- point
 - en géométrie algorithmique, 901, 902
- point-valeur
 - représentation par, 798
- pointeur, 18
 - implémentation par tableau, 203–208
- polaire (angle)
 - tri de points en fonction de leur, 907
- Pollard (heuristique rho de), 865–870
- POLLARD-RHO, 866
- polygone, 907
 - calcul de la surface d'un, 908
 - convexe, 907
 - déterminer si deux polygones simples se croisent, 914
 - en forme d'étoile, 924
 - noyau d'un, 924
 - test de simplicité, 914
- polynôme, 49, 795
 - comportement asymptotique, 55
 - évaluation d'un, 798, 802, 818
 - interpolation par un, 798
 - représentation par paires point-valeur d'un, 798

- polynômes
 addition des, 796
 multiplication des, 796, 801, 802, 816
 représentation par coefficients des, 797
- polynomial
 schéma d'approximation en, 988
- polynomial (temps)
 acceptation en, 944
 algorithme à, 933
 calculabilité en, 942
 décision en, 944
 réductibilité (\leq_P) en, 951
 résolubilité en, 941
 schéma d'approximation en, 988
 vérification en, 946–950
- polynomialement (reliés), 942
- pondération (fonction de)
 dans un matroïde pondéré dans un matroïde pondéré, 385
 pour un graphe, 515
- pondéré
 arbre équilibré, 416
 matroïde,
 matroïde matroïde, 385
 médián, 188
- pondérée
 problème de la couverture d'ensemble, 1014
 heuristique de l'union pondérée, 492
- pont, 542
- porte, 954
- positif (flot), 628
- postfixe (parcours), 249
- potentiel
 contrainte de, 583–589
 d'une structure de données, 403
 fonction, 403
 graphe de,
- potentiel (méthode du), 402–406
 pour algorithme pousser-réétiqueter générique,
 pour algorithme préflot générique,
 pour l'algorithme de Knuth-Morris-Pratt, 895
 pour les compteurs binaires, 404, 405
 pour les opérations de pile, 404
 pour les tables dynamiques, 409, 410, 412–414
 pour les tas, 406
 pour les tas de Fibonacci, 468–471, 476, 477, 479–481
 pour structure de données d'ensembles disjoints, 500–505
- pour**, 17
 en pseudo code,
 et invariants de boucle, 16
- poussage (opération de)
 dans un algorithme de préflot, 651
- POUSSER, 651
 opération préflot, 651
- PP, 526
- ppcm (plus petit commun multiple), 833
- Pr { } (distribution de probabilité), 1063
- préflot (algorithmes),
 algorithme générique,
 algorithme réétiqueter-vers-l'avant,
 avec file de sommets débordants, 669
 opérations élémentaires,
- pré-tri, 929
- PRÉDÉCESSEUR, 192
- prédécesseur
 dans un arbre binaire de recherche, 252, 253
 dans un arbre de parcours en largeur, 518
 dans un arbre de plus court chemin, 567
 dans un arbre de rangs, 303
 dans un arbre rouge-noir, 270
 dans un B-arbre, 437
 dans une liste chaînée dans une liste chaînée, 199
- prédécesseur (sous-graphe de)
 dans un parcours en largeur, 523
- prédécesseur (sous-graphe)
 dans un parcours en profondeur, 525
 dans un plus court chemin à origine unique, 567
- préemption, 392
- préfixe
 codage, 377
 d'une chaîné d'une chaîne (\sqsubset), 877
 d'une séquence, 343
- préfixe (fonction), 892, 893
 lemme de l'itération, 896
- PRÉFLOT-GÉNÉRIQUE, 653
- premier ancêtre commun, 508
- premiers (nombres), 823
 densité des, 857
 fonction de distribution des, 857
 théorème des, 857

- Prim (algorithme de)
 - avec matrice d'adjacences, 557
 - avec poids entiers, 557
 - implémenté à l'aide d'un tas, 556
 - implémenté à l'aide d'un tas de Fibonacci, 556
 - similitude avec algorithme de Dijkstra, 554, 582
- primaire (mémoire), 426
- primarité
 - relative, 826
 - test de, 856–865, 872
 - test de Miller-Rabin, 859–865
 - test de pseudo primarité, 858, 859
- primitive (racine de \mathbf{Z}_n^*), 847
- principale (mémoire), 426
- principe d'inclusion et d'exclusion, 1038
- principe du 0-1, 692, 694
- priorité (file de)
 - dans l'algorithme de Prim, 554, 556
 - pour la construction des codages de Huffman, 378
- probabiliste (analyse)
 - borne de la taille d'une alvéole pour chaînage, 243
 - d'une enveloppe convexe dans une distribution à enveloppes non denses, 931
 - de l'adressage ouvert, 235–238
 - de l'heuristique rho de Pollard, 867–870
 - de l'insertion dans un ABR ayant des clés égales, 262
 - de la borne du plus long sondage pour hachage, 243
 - de la comparaison de fichiers, 885
 - de la hauteur d'un ABR construit aléatoirement, 258–261
 - de la profondeur moyenne d'un nœud dans un arbre binaire de recherche construit aléatoirement, 263
 - des collisions, 222
 - des minorants pour le cas moyen du tri, 171
 - du hachage avec chaînage du hachage avec chaînage, 220–222
 - du hachage par adressage ouvert, 238
 - du hachage universel, 227–229
 - du parcours d'une liste compacte, 212
 - du partitionnement, 147, 153, 156
 - du test de primarité de Miller-Rabin, 862–865
 - du tri par paquet, 171
 - du tri rapide, 156, 261
 - tri de points selon la distance par rapport à l'origine, 171
- probabilité, 1063–1069
 - distribution de, 1063, 1064
 - fonction de distribution de, 171
- problème
 - abstrait, 939
 - concret, 940
 - d'optimisation, 940
 - de décision, 940
 - de la couverture d'ensemble pondérée, 1014
 - de la recherche, 18
 - du sac-à-dos fractionnaire, 373
 - intraitable, 933
 - optimisation, 315
 - solution d'un, 940
 - solution de, 940
 - traitable, 933
 - problème d'optimisation, 936
 - et problème de décision, 936
 - problème de décision, 936
 - et problème d'optimisation, 936
 - problème de l'embauche, analyse probabiliste du, 93, 94
 - en ligne, 110–112
 - problème de l'ordonnancement pour machines parallèles, 1015
 - problème de la programmation linéaire entière, 982
 - problème de la réalisabilité d'inégalités linéaires, 791
 - problème de la somme de sous-ensemble NP-complétude du, 933
 - problème de programmation entière zéro-un, 1004
 - problème de programmation linéaire entier, 752, 792
 - problème de réalisabilité, 791
 - problème du collecteur de coupons, 105
 - problème du vestiaire à chapeaux, 94
 - problème MAX-CUT, 1007
 - procédure, 14
 - proche (heuristique du point le plus), 997
 - produit, 705
 - cartésien, 1037
 - croisé, 903

- de matrices, 705
- de polynômes, 796
- extérieur, 706
- règle du, 1058
- scalaire, 706
- scalaire-flot, 632
- profondeur
 - arbre de parcours en, 525
 - d'un nœud dans un arbre enraciné, 1051
 - d'un nœud dans un arbre rouge-noir, 272, 304
 - d'un réseau de comparaison, 684
 - d'un réseau de tri, 685
 - d'une feuille dans un arbre de décision, 161
 - d'une pile, 155
 - de l'arbre récursif du tri rapide, 147
 - de TRIEUSE, 696
 - forêt de parcours en, 525
 - moyenne, d'un nœud dans un arbre binaire de recherche construit aléatoirement, 263
 - recherche de la, 507
- profondeur (parcours en)
 - d'un arbre, 249
- programmation dynamique
 - comparée à algorithmes gloutons, 365–367
- programmation linéaire, 745–794
 - voir aussi* problème de programmation linéaire entier, problème de programmation entière 0-1, 745
 - algorithme de Karmarkar, 752, 794
 - algorithme du simplexe, 765–779
 - algorithmes, 752
 - applications, 751, 752
 - dualité, 779–785
 - et flot à coût minimal, 762, 763
 - et flot maximum, 761
 - et flot multi-ressources, 763, 764
 - et flot multi-ressources à coût minimal, 765
 - et plus court chemin à couple unique, 761
 - forme canonique, 754
 - forme relâchée, 756–759
 - forme standard, 756
 - méthode du point intérieur, 752, 794
 - recherche d'une solution initiale, 785–790
 - théorème fondamental, 790
- programme linéaire auxiliaire, 785
- programme linéaire de maximisation, 748
 - et programmes linéaires de minimisation, 754
- programme linéaire de minimisation, 748
 - et programmes linéaires de maximisation, 754
- programme linéaire dual, 780
- programme linéaire irréalisable, 754
- programme linéaire non borné, 754
- programme linéaire primal, 780
- programme linéaire réalisable, 754
- programmes linéaires équivalents, 754
- propre
 - ancêtre, 1050
 - descendant, 1050
 - sous-ensemble (\subset), 1034
 - sous-groupe, 837
- propriété aucun-chemin, 570, 590
- propriété de convergence, 570, 590, 591
- propriété de majorant, 570, 590
- propriété de relâchement de chemin, 570, 591
- propriété de sous-graphe prédécesseur, 570, 593, 594
- propriété de tas max, 123
- propriété de tas min, 123, 449
 - conservation, 126
- pseudo code, 14, 17, 18
 - bloc, 17
 - boucles, 17
 - commentaire (\triangleright), 17
 - indentation, 17
 - variable, 17
- pseudo premier, 858, 859
- pseudo-inverse, 739
- PSEUDO-PREMIER, 858
- publique (clé), 851, 854
- publiques (cryptosystème à clés), 850–856
- puissance
 - d'un élément modulo n , 846–850
- puissance k ième, 827
- puissance non triviale, 827
- puits, 626, 629
- pure (séquence), 689
- quadratique
 - résidu, 871
 - sondage, 233, 234, 244
- quantile, 186
- queue

- d'une distribution binomiale, 1079–1085
- d'une file, 197
- d'une liste chaînée, 199
- quicksort, 139–157
- Quicksort-Randomise, 261
 - et arbres binaires de recherche construits aléatoirement, 263
- quotient, 824
- R** (ensemble des nombres réels), 1034
- RABIN-KARP, 883
- Rabin-Karp (algorithme de), 880–885
- racine
 - d'une arborescence, 1050
 - de Z_n^* , 847
 - principale de l'unité, 804
- racine (liste des)
 - d'un tas binomial, 449
 - d'un tas de Fibonacci, 468
- RAM *voir* machine à accès aléatoire, 19
- RANDOM, 90
- RANDOMISATION-DIRECTE, 98
- randomisé (algorithme)
 - de partitionnement, 153, 156
 - hachage universel, 226–229
 - heuristique rho de Pollard, 865–870
 - performances dans le cas le plus défavorable, 148
 - pour l'insertion dans un ABR ayant des clés égales, 262
 - pour le tri rapide, 147, 148, 153, 156
 - pour sélection, 179–183
 - test de primarité de Miller-Rabin, 859–865
- rang
 - colonne, 707
 - d'un nœud dans une forêt d'ensembles disjoints, 495, 500, 505
 - d'un nombre dans un ensemble trié, 296
 - d'une matrice, 707
 - dans un arbre de rangs, 298, 299, 301
 - ligne, 707
 - plein, 707
 - statistique, 177–189
- rangs
 - dynamiques, 296–301
- rangs (arbre de), 296–301
- rapide (tri)
 - analyse du, 143–153
 - analyse du cas le plus défavorable, 149
 - avec la méthode du médian parmi 3, 156
 - bonne implémentation du cas le plus défavorable, 186
 - description du, 139–143
 - profondeur de pile du, 155
 - utilisation du tri par insertion dans le, 153
 - version récursive terminale du, 155
 - versions randomisés du, 147, 148
- rapprochés (recherche des deux points les plus), 925–929
- réétiquetage (opération de), 652, 656
- réalisable (solution), 584
- recensement, 339–341
- RECHERCHE-INTERVALLE, 307
- recherche
 - d'un intervalle exact, 310
 - dans tableau non trié, 114
 - dans un arbre binaire de recherche, 250, 251
 - dans un arbre d'intervalles, 306–309
 - dans un arbre rouge-noir, 270
 - dans un B-arbre, 432, 433
 - dans une liste compacte, 212
 - dans une table à adressage direct, 217
 - dans une table à adressage ouvert, 232
 - dans une table de hachage chaînée, 219
 - de chaîne de caractères, 875–900
 - dichotomique, 34
 - linéaire, 18
- recherche (arbre binaires de)
 - propriété, 248
- recherche (arbre de) *voir* équilibré (arbre de recherche), binaire de recherche (arbre), B-arbre, intervalles (arbre d'), rangs (arbre de), rouge-noir (arbre), déployé (arbre), 2-3 (arbre), 2-3-4 (arbre), 247
- recherche dichotomique
 - dans tri par insertion, 34
 - méthode diviser pour régner, 34
- RECHERCHE-ALÉATOIRE, 114
- RECHERCHE-AUTOMATE-FINI, 888
- RECHERCHE-AVEC-RÉPÉTITION, 900
- RECHERCHE-DÉTERMINISTE, 114
- RECHERCHE-LISTE, 200
- RECHERCHE-LISTE', 202
- RECHERCHE-LISTE-COMPACT, 212
- RECHERCHE-LISTE-COMPACTE', 213
- RECHERCHE-MÉLANGE, 114
- RECHERCHE-NAÏVE, 878
- RECHERCHE, 192

- RECHERCHER-ADRESSAGE-DIRECT, 217
 RECHERCHER-ARBRE, 433
 RECHERCHER-B-ARBRE, 432, 438
 RECHERCHER-HACHAGE, 232, 238
 RECHERCHER-HACHAGE-CHAÎNÉE, 219
 RECHERCHER-INTERVALLE, 309
 RECHERCHER-INTERVALLE-EXACT, 310
 RECHERCHER-RANG-CLÉ, 301
 reconstruction d'une solution optimale en programmation dynamique, 338, 339
 recoupement (intervalles)
 point de recoupement maximal, 310
 recouvrement des suffixes (lemme de), 877
 rectangles superposés, 310
 récupération de la mémoire (*garbage collector*), 121
 récupération de place mémoire, 205
 RÉCUPÉRER-CHAÎNE, 339
 RÉCUPÉRER-RANG, 297, 301
 récurrence, 59–85
 méthode diviser-pour-régner et, 59–85
 résolution par la méthode itérative,
 résolution via méthode de l'arbre récursif,
 résolution via méthode de substitution, 60–64
 résolution via méthode générale,
 solution via méthode de Akra-Bazzi, 84, 85
 récursif (arbre)
 du tri par fusion, 341
 récursivité terminale, 369
 réductibilité, 951–953
 réduction
 algorithme de, 951
 fonction de, 951
 réétiqueté (sommet), 652
 RÉÉTIQUETER, 652
 RÉÉTIQUETER-VERS-L'AVANT, 665
 réétiqueter-vers-l'avant (algorithme), 659–669
 réflexive (relation), 1038
 région de réalisabilité, 749
 région, de réalisabilité, 749
 règle
 de la somme, 1058
 du produit, 1058
 règle de Horner, 36
 régulier (graphe d -), 648
 RÉINITIALISER, 402
 rejet
 par un algorithme, 944
 par un automate fini, 886
 relâché (tas), 485
 relâchement,
 d'un arc, 568–570
 RELÂCHER, 569
 relation, 1038–1040
 relaxation
 programmation linéaire, 1005
 RELIER-TAS-FIB, 475
 remplissage (facteur de)
 d'une table de hachage, 220
 d'une table dynamique, 407
 repère, 132
 répétées (élevations au carré)
 pour calculer la puissance d'un nombre, 848
répéter, en pseudo code, 17
 répétition (facteur de)
 d'une chaîne, 899
 repondération
 des plus courts chemins à origine unique, 596
 et plus courts chemins pour tout couple de sommets, 616
 représentant d'un ensemble, 487
 représentation
 par listes d'adjacences, 514
 par matrice d'adjacences, 515
 requête, 192
 réseau, 681
 admissible, 660, 661
 de comparaison, 682–686
 de fusion pair-impair, 697
 de permutation, 698
 de transposition, 697
 de tri bitonique, 689–692
 de tri pair-impair, 697
 pour la fusion, 692–694
 résiduel, 633–635
 réseau de tri, 681–699
 AKS, 699
 basé sur le tri par fusion, 694–696
 basé sur le tri par insertion, 685
 bitonique, 689–692
 pair-impair, 697
 résidu, 871
 résiduel
 arc, 633

- réseau, 633–635
- résiduelle (capacité), 633, 635
- RÉSOLUTION-EQUATIONS-LINÉAIRES-MODULAIRES, 841
- RÉSOLUTION-LUP, 721
- reste, 49, 824
- rho (heuristique), 865–870
- right-convert, 273
- RN-ENUMÉRER, 304
- RN-INSÉRER, 273
- RN-INSÉRER-CORRECTION, 274
- RN-JOINTURE, 288
- RN-SUPPRIMER, 281
- RN-SUPPRIMER-CORRECTION, 282
- rotation
 - circulaire, 899
 - dans un arbre rouge-noir, 271–273
- ROTATION-DROITE, 272
- ROTATION-GAUCHE, 272, 309
- rouge-noir (arbre)
 - comparaison avec les B-arbres, 431
 - et arbres 2-3-4, 431
 - pour déterminer si des segments de droite sont sécants, 911
- RSA (cryptosystème à clés publiques), 850–856
- sac-à-dos (problème du)
 - 0-1, 946
 - fractionnaire, 373
 - variante du tout ou rien, 373
 - variante entière, 375
 - variante fractionnaire, 375
- SAT, 962
- satellite (données), 117
- satellites (données), 192
- satisfaisabilité, 955, 962–964, 1003, 1004, 1007
 - satisfaisabilité 2-CNF, 968
 - et satisfaisabilité 3-CNF, 934
 - satisfaisabilité 3-CNF, 965–968
 - et satisfaisabilité 2-CNF, 934
 - satisfaisabilité CNF, 1007
 - satisfaisabilité MAX-3-CNF, 1003, 1004
 - algorithme d'approximation pour, 1003, 1004
 - satisfaisabilité MAX-CNF, 1007
- satisfaisable (formule), 962
- saturant (poussage), 651, 656
- saturé (arc), 651
- scalaire
 - produit, 706
- scalaire (produit), 705
- scalaire-flot (produit), 632
- schéma d'approximation, 988
- Schur
 - complément de, 723, 736
 - lemme du complément de, 736
- scission d'arbres 2-3-4, 443
- secondaire (espace de stockage)
 - arbre de recherche pour un, 425–444
 - pires sur, 442
- secrète (clé), 851, 854
- segment *voir* orienté (segment), droite (segment de), 902
- SÉLECTION, 183, 184
- sélection
 - dans un arbre de rangs, 297, 298
 - en temps attendu linéaire, 179–183
 - en temps linéaire dans le cas le plus défavorable, 183–187
 - et tri par comparaison, 186
 - problème de, 177
- SÉLECTION-RANDOMISÉE, 180
- semi satisfaisabilité 3-CNF, 983
- semi-anneau fermé, 623
- semi-connexe (graphe), 542
- sentinelle, 27, 201, 202, 268
- SÉPARATEUR, 689
- séquence, 682, 689
 - bitonique, 599
 - inversion dans, 36
- séquence bitonique
 - et plus courts chemins, 599
- séquence monotone, 136
- série, 81, 1023
 - formelle de puissances, 81
- si**, en pseudo code, 17
- simple
 - graphe, 1044
- simple (liste chaînée)[*voir aussi* chaînée (liste)], 199
- simple (polygone)
 - détermination d'un, 914
- SIMPLEXE, 772
- simplexe, 750
- singleton, 1036
- singulière (matrice), 706
- solution

- d'un problème concret, 940
- d'un système d'équations linéaires, 717
- de base, 767
- de problème abstrait, 940
- irréalisable, 753
- optimale, 753
- réalisable, 584, 749, 753
- solution de base, 767
- solution irréalisable, 753
- solution optimale, 753
- solution réalisable, 749, 753
- solution réalisable de base, 767
- sommation, 1021–1032
 - bornes sur les,
 - dans notation asymptotique, 45
 - et notation asymptotique, 1022
 - formules et propriétés des, 1022–1025
 - implicite, 630
 - lemme de la, 805
 - linéarité des, 1022
- somme
 - cartésienne, 803
 - de polynômes, 796
 - flot, 632
 - infinie, 1022
 - règle de la, 1058
- somme d'un sous-ensemble (problème de la)
 - avec cible unaire, 982
- somme de sous-ensemble (problème de la)
 - algorithme d'approximation pour, 1007–1013
- somme emboîtée, 1024
- SOMME-EXACTE-SOUS-ENSEMBLE, 1008
- SOMME-SOUS-ENSEMBLE, 979
- SOMME-SOUS-ENSEMBLE-APPROCHÉE, 1010
- SOMMET, 918
- sommet
 - d'un polygone, 907
 - d'une pile, 196
 - intermédiaire, 610
 - isolé, 1044
 - point d'articulation, 542
- sommet (couverture de), 972, 989
 - dans un arbre, 992
- sommet sélecteur, 975
- sommets (ensemble des), 1043
- sondage, 243
- sonde, 231
- sortant (degré), 1044
- sortie
 - câble de, 682
 - d'une porte logique, 954
 - de circuit combinatoire, 955
 - séquence de, 682
- source, 626, 629
- SOURCE-UNIQUE-INITIALISATION, 568
- sous-arborescence, 1050
- sous-arbre
 - conservation de la taille d'un, dans un arbre de rangs, 299, 300
- sous-chaîne, 1058
- sous-chemin, 1044
- sous-déterminé (système d'équations linéaires), 718
- sous-ensemble (\subseteq), 1034
- sous-ensembles
 - famille héréditaire de, 384
 - famille indépendante de, 384
- sous-graphe, 1045
 - isomorphe (problème du), 982
- sous-groupe, 837–839
- sous-routine
 - appel, 18, 19, 22
 - exécution, 22
- sous-séquence, 342
 - commune, 342
 - commune (plus longue), 341–347
- SOUS-SOMMET, 918
- sous-structure optimale
 - d'arbres binaires de recherche, 350
 - d'un ordonnancement de chaîne de montage, 317–319
 - d'un plus court chemin, 565
 - de plus courts chemins non pondérés, 334
 - du choix d'activités, 363, 364
- spline, 742
- stabilité
 - des algorithmes de tri, 164, 167
 - numérique, 701, 743
- Stirling
 - formule de, 52
- stockage (gestion du), 205–208, 223
- Strassen (algorithme de), 710–717
- structure de données, 191
 - arbre AA, 293
 - arbre AVL, 289
 - arbre de fusion, 175, 423
 - arbre de recherche exponentiel, 175, 423

- arbres bouc-émissaire, 293
- arbres équilibrés en poids, 293
- arbre k -voisin, 293
- de van Emde Boas, 136, 423
- ensemble dynamique, 191
- file de priorité, 131–134
- listes à saut, 293
- pour graphe dynamique, 423
- tas, 121–137
- treap, 289
- succès dans une épreuve de Bernoulli, 1074
- SUCCESEUR, 192
- successeur
 - dans un arbre binaire de recherche, 252, 253
 - dans un arbre de rangs, 303
 - dans un arbre rouge-noir, 270
 - dans une liste chaînée, 199
- suffixe (\square), 877
- suffixe (fonction), 887
 - lemme de l'inégalité, 889
 - lemme de la récursivité, 889
- suite ($\langle \rangle$), 1041
 - inversion dans, 94
- suites, 106–110
- supérieure (matrice triangulaire), 703
- superposés (intervalles)
 - trouver tous les, 309
- superpuits, 629
- supersource, 629
- SUPPRESSION, 192
- suppression
 - d'un état de droite de balayage, 910
 - d'une pile, 195, 197
 - dans tas, 134
 - dans un arbre binaire de recherche, 255–257
 - dans un arbre d'intervalles, 306
 - dans un arbre de rangs, 300
 - dans un arbre rouge-noir, 281–286
 - dans un B-arbre, 439–442
 - dans un tas binomial, 460, 461
 - dans un tas de Fibonacci, 481, 484
 - dans un tas 2-3-4, 462
 - dans une liste chaînée dans une liste chaînée, 200, 201
 - dans une table à adressage direct, 217
 - dans une table à adressage ouvert, 232
 - dans une table de hachage chaînée, 219
 - dans une table dynamique, 412
- SUPPRIMER, 445
- SUPPRIMER-ADRESSAGE-DIRECT, 217
- SUPPRIMER-B-ARBRE, 439
- SUPPRIMER-HACHAGE, 238
- SUPPRIMER-HACHAGE-CHAÎNÉE, 219
- SUPPRIMER-INTERVALLE, 305
- SUPPRIMER-LISTE, 201
- SUPPRIMER-LISTE', 201
- SUPPRIMER-MOITIÉ-SUPÉRIEURE, 406
- SUPPRIMER-PISANO, 484
- Supprimer-Tas, 134
- SUPPRIMER-TAS-FIB, 481
- suprapolynomial (temps), 933
- SUR-SEGMENT, 905
- surdéterminé (système d'équations linéaires), 718
- sûre (arête), 547
- surface d'un polygone simple, 908
- surjection, 1041
- SVD, 744
- symboles (table des), 215, 224
- symétrie, 626
- symétrique
 - dans un groupe, 834
 - relation, 1038
- symétrique (matrice), 704, 709
 - définie positive, 735–737
- systèmes
 - d'équations linéaires, 717–730, 741
 - de contraintes de potentiel, 583–589
- table de hachage
 - secondaire, 239
- table de hachage à adressage ouvert
 - double hachage, 238
- table des symboles, 226
- TABLE-SUPPRIMER, 412
- tableau, 17
 - de Monge, 83
 - de Young, 135
- tâches (ordonnancement de), 389–392, 394
- taille
 - d'un arbre binomial, 448
 - d'un ensemble, 1036
 - d'un réseau de comparaison, 685
 - d'un sous-arbre dans un tas de Fibonacci, 483
 - d'une clique, 969
 - d'une couverture de sommet, 972

- d'une couverture de sommets, 989
- de l'entrée d'un algorithme, 21, 822, 940–943
- tant que**, en pseudo code, 17
- Tarjan (algorithme différé pour le premier ancêtre commun), 508
- tas, 121–137
 - analysé par la méthode du potentiel, 406
 - augmentation d'une clé dans, 132, 133
 - binaire *voir* tas, 121
 - binomial *voir* binomial (tas), 445
 - clé maximale, 132
 - conservation de la structure de, 124–126
 - construction, 126–129, 135
 - d*-aire, 135, 621
 - dans algorithme de Dijkstra, 581
 - dans algorithme de Johnson, 620
 - dans l'algorithme de Prim, 556
 - de Fibonacci *voir* Fibonacci (tas de), 465
 - 2-3-4, 462
 - en tant que mémoire récupérable, 121
 - et file de priorité, 131–134
 - et treap, 289
 - extraction de la clé maximale, 132
 - fusionnable *voir* fusionnable (tas), 445
 - hauteur, 123
 - insertion dans, 133, 134
 - relâché, 485
 - suppression dans, 134
 - temps d'exécution des opérations sur un, 446
- tas (propriété de), 123
 - comparée à la propriété des ABR, 250
- tas de Fibonacci
 - dans algorithme de Johnson, 620
- tas max, 123
 - augmentation d'une clé dans, 132, 133
 - clé maximale, 132
 - et file de priorité, 131–134
 - extraction de la clé maximale, 132
 - fusionnable, 211, 421, 445
 - insertion dans, 133, 134
 - suppression dans, 134
- tas min, 123
 - dans algorithme de Dijkstra, 581
 - dans algorithme de Johnson, 620
 - et file de priorité min, 134
 - fusionnable, 211, 421
- TAS-BINOMIAL-DIMINUER-CLÉ, 458
- TAS-BINOMIAL-EXTRAIRE-MIN, 458
- TAS-BINOMIAL-INSÉRER, 457, 462
- TAS-BINOMIAL-SUPPRIMER, 460, 461
- TAS-DIMINUER-CLÉ, 134
- TAS-EXTRAIRE-MIN, 134
- TAS-FIB-MODIFIER-CLÉ, 485
- TAS-MINIMUM, 134
- TAS-SUPPRIMER, 134
- tautologie, 950, 968
- taux de croissance, 24
- Taylor (série de), 264
- TÉMOIN, 860
- témoin du caractère composite d'un nombre, 859
- temps *voir* temps d'exécution, 21
- temps d'exécution, 21
 - attendu, 24
 - cas le plus défavorable, 23
 - cas le plus favorable, 25
 - du cas le plus défavorable, 44
 - moyen, 24
 - optimal, 44
 - ordre de grandeur, 24
 - taux de croissance, 24
- terminal
 - appel récursif, 155
- tête
 - d'une file, 197
 - d'une liste chaînée, 199
- texte dans une recherche de chaîne de caractères, 875
- Théorème fondamental de la programmation linéaire, 790
- théorème général, 70
 - démonstration, 72–80
- Toeplitz (matrice de), 817
- topologique (tri), 534–536
 - lors du calcul du plus court chemin à origine unique sur un graphe orienté sans circuit, 575
- total (ordre), 1040
- tournée
 - bitonique, 354
 - d'Euler, 934
 - dans un graphe, 978
- tournée d'Euler, 934
 - et cycle hamiltonien, 934
- TOURNÉE-VC-APPROCHÉE, 993
- traitabilité, 933
- transformée rapide de Fourier (FFT), 795

- circuit pour la, 815, 816
- discrète, 803
- et arithmétique modulaire, 819
- implémentation itérative de la, 811–814
- implémentation récursive de la, 807
- multidimensionnelle, 817
- transition (fonction de), 886, 890, 891, 899
- transitive (fermeture), 613–616
 - d'un graphe dynamique, 621
 - et multiplication des matrices booléennes, 734
- transitive (relation), 1038
- transparence de cache, 444
- transport (réseau de),
 - avec capacités négatives, 672
 - correspondant à un graphe biparti, 645
 - coupure dans un, 636–638
- transposée
 - conjuguée, 734
 - d'un graphe orienté, 516
 - d'une matrice, 515, 702
- transposition
 - réseau de, 697
- transverse
 - arc, 531
- traversée, 904
- traversée d'une coupe, 547
- tri, 13–17, 26–34, 117–175
 - à bulles, 35
 - à l'aide d'un arbre binaire de recherche, 257
 - à l'aide d'un réseau[voir réseau de tri], 681
 - d'éléments de longueur variable, 173
 - de points en fonction de leur angle polaire, 907
 - de Shell, 37
 - en temps linéaire, 162–172
 - flou, 156
 - lexicographique, 262
 - méthode diviser pour régner, 26–34
 - minorant pour le, 159–162
 - minorant pour le cas moyen du, 171
 - par base, 164–167
 - par dénombrement, 162–164
 - par fusion, 26–34
 - par insertion, 13–17, 22, 23
 - par paquet,
 - par paquets,
 - par sélection, 25
 - par tas, 121–137
 - problème du, 117
 - rapide, 139–157
 - sur place, 15, 119
 - topologique, 534–536
- tri par base
 - comparé au tri rapide, 166, 167
- tri par comparaison
 - et sélection, 186
- tri par fusion
 - tri par insertion dans, 35
- tri par insertion
 - arbre de décision, 160
 - dans le tri par paquet,
 - dans le tri par paquets,
 - dans tri par fusion, 35
 - via recherche dichotomique, 34
- tri rapide
 - analyse du cas moyen, 149–152
 - comparé au tri par base, 166, 167
 - version randomisée, 154
- TRI-BASE, 166
- TRI-BULLES, 35
- TRI-DÉNOMBREMENT, 162
- TRI-FAIRE-VALOIR, 155
- TRI-FUSION, 30
 - arbre récursif du, 341
- TRI-INSERTION, 15, 22
- TRI-PAR-PAQUETS, 168
- TRI-PAR-TAS, 129
- TRI-RAPIDE, 140
- TRI-RAPIDE', 155
- TRI-RAPIDE-RANDOMISE, 148
- TRI-TOPOLOGIQUE, 535
- triangle de Pascal, 1062
- triangulaire
 - inégalité, 992
 - matrice, 703, 704, 709
- trichotomie, 47
- tridiagonal (système linéaire), 741
- tridiagonale (matrice), 703
- trié en tas min, 449
- triée (liste chaînée)[voir aussi chaînée (liste)], 199
- triée (liste chaînée)triée (liste chaînée), 199
- TRIEUSE, 695, 696
- TRIEUSE-BITONIQUE, 690
- triviaux (diviseurs), 823
- trois (forme normale conjonctive d'ordre), 965

- 3-COULEURS, 984
 trou noir, 517
 TROUVER-ENSEMBLE, 488
 implémentation d'une forêt d'ensembles
 disjoints, 497, 509
 implémentation en liste chaînée de, 491, 493
 TROUVER-PROFONDEUR, 507
 type de données
 entier, 19
 virgule flottante, 19
- unaire, 941
 uniforme (distribution de probabilité), 1064
 uniforme (hachage), 232
 simple, 221
 UNION, 203, 445, 488
 implémentation d'une forêt d'ensembles
 disjoints, 496
 implémentation en liste chaînée, 493
 union
 d'ensembles (\cup), 1034
 de deux tas, 445
 de deux tas binomiaux, 452–457
 union de deux tas 2-3-4, 462
 de langages, 943
 de listes chaînées, 203
 par rang, 495
 UNION-TAS-FIB, 471
 unique (décomposition)
 en facteurs premiers, 827
 unique (plus court chemin à origine), 563–599
 algorithme de Bellman-Ford, 571–575
 algorithme de changement d'échelle de Gabow, 596
 algorithme de Dijkstra, 577–583
 dans un graphe orienté acyclique, 575–577
 et contraintes de potentiel, 583–589
 unique (plus court chemin avec origine)
 dans un graphe ε -dense, 621
 unitaire (matrice triangulaire), 703, 704
 unité (vecteur), 702
 univers, 1035
 universel (hachage), 226–229
- valeur
 d'un flot, 627
 d'une fonction, 1041
 objectif, 749, 753
 valeur attendue
 d'une variable aléatoire indicatrice, 91
 valeur de l'objectif, 753
 optimale, 754
 valeur de l'objectif optimale, 754
 valeur objectif, 749
 valide (décalage), 875
 valué (graphe bipartite)
 couplage d'un, 485
 Vandermonde (matrice de), 710
 variable
 voir aussi variable aléatoire indicatrice, 91
 aléatoire, 1069–1074
 d'écart, 757
 de base, 758
 entrante, 768
 globale, 17
 locale, 17
 non de base, 758
 sortante, 768
 variable aléatoire
 indicatrice *voir* variable aléatoire indica-
 trice, 91
 variable aléatoire indicatrice, 91–94
 dans algorithme d'approximation pour sa-
 tisfaisabilité MAX-3-CNF, 1003, 1004
 dans analyse de l'insertion dans un arbre-
 tas, 289
 dans analyse de la hauteur attendue d'un
 arbre binaire de recherche construit
 aléatoirement, 258–261
 dans analyse de la sélection randomisée,
 183
 dans analyse de suites, 109, 110
 dans analyse du hachage, 221, 222
 dans analyse du hachage universel, 227,
 228
 dans analyse du problème de l'embauche,
 93, 94
 dans analyse du tri par paquets, 167
 dans analyse du tri rapide, 150–152, 154
 dans bornage de la queue de droite de la
 distribution binomiale, 1082, 1083
 variable d'écart, 757
 variable de base, 758
 variable entrante, 768
 variable hors-base, 758
 variable sortante, 768
 Var [] (variance), 1072
 variance, 1072

- d'une distribution binomiale, 1077
- d'une distribution géométrique, 1075
- VC, 978
- vecteur, 702, 706, 707
 - du plan, 902
 - orthonormal, 744
 - produit en croix de, 903
- vecteur d'annulation, 707
- vecteurs (convolution de), 798
- Venn (diagramme de), 1035
- vérification, 946–950
 - d'arbre couvrant, 561
- vérité (table de), 954
- vide
 - chaîne, 943
 - chaîne (ε), 877
 - langage (\emptyset), 943
- vide (ensemble)
 - lois pour, 1034
- violation d'une contrainte d'égalité, 766
- VISITER-PP, 526
- Viterbi (algorithme de), 358
- VLSI (very-large-scale integration), 82
- voisin, 1046
- voisinage, 648
 - liste de, 661
- voyageur de commerce (problème du)
 - algorithme d'approximation pour, 992–997
 - avec goulot d'étranglement, 997
 - avec inégalité triangulaire, 993–995
 - bitonique, 354
 - NP-complétude du, 978, 979
 - sans inégalité triangulaire, 995, 996
- Yen (amélioration de)
 - pour l'algorithme de Bellman-Ford, 595
- Z** (ensemble des entiers), 1034
- \mathbf{Z}_n (classes d'équivalence modulo n), 824
- \mathbf{Z}_n^* (éléments du groupe multiplicatif modulo n), 835
- \mathbf{Z}_n^+ (éléments non nuls de \mathbf{Z}_n^*), 858
- zéro (matrice), 702
- zéro d'un polynôme modulo un nombre premier, 843
- 0-1 (principe du), 686–689
- 0-1 (problème de la programmation entière), 982
- 0-1 (problème du sac-à-dos), 946