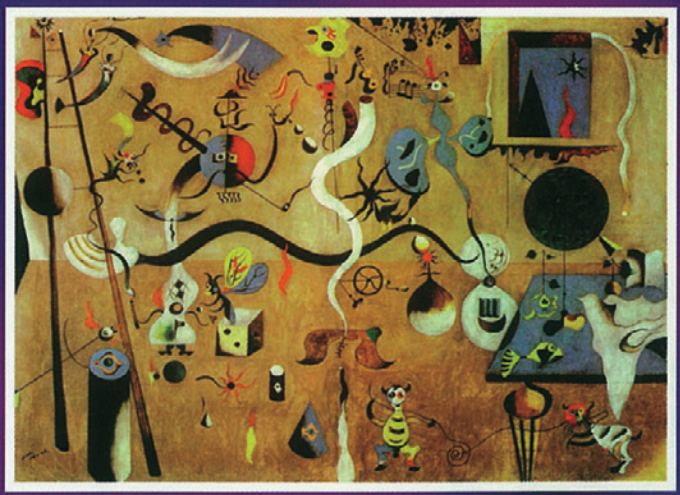
PROJETO DE ALGORITMOS

Fundamentos, análise e exemplos da Internet

MICHAEL T. GOODRICH / ROBERTO TAMASSIA



bookman®

CÓPIA NÃO AUTORIZADA É CRIME
ABDR
ASSOCIAÇÃO BRASILEIRA DE DIREITOS REPROGRÁFICOS
RESPEITE O DIREITO AUTORAL

G655p Goodrich, Michael T.

Projeto de algoritmos [recurso eletrônico] : fundamentos, análise e exemplos da Internet / Michael T. Goodrich e Roberto Tamassia ; tradução Bernardo Copstein e João Batista Oliveira. – Dados eletrônicos. – Porto Alegre : Bookman, 2008.

Editado também como livro impresso em 2004.
ISBN 978-85-7780-342-2

1. Ciência da computação – Algoritmos – Projeto. I. Tammassia, Roberto. II. Título.

CDU 004.421/.6

Catalogação na publicação: Renata de Souza Borges CRB-10/Prov-021/08

MICHAEL T. GOODRICH
Departamento de Informação e Ciência da Computação
University of California, Irvine

ROBERTO TAMASSIA
Departamento de Ciência da Computação
Brown University

PROJETO DE ALGORITMOS

Fundamentos, análise e exemplos da Internet

Tradução:

Dr. Bernardo Copstein
Doutorado na área de simulação de sistemas por computador pelo Programa de Pós-Graduação em Ciência da Computação da UFRGS
Professor da Faculdade de Informática da PUCRS

Dr. João Batista Oliveira
Doutorado na área de matemática computacional e algoritmos pela Technische Universität Hamburg-Harburg, de Hamburgo, Alemanha
Professor da Faculdade de Informática da PUCRS

Versão impressa
desta obra: 2004

bookman®

2008

Obra originalmente publicada sob o título
Algorithm Design: Foundations, Analysis, and Internet Examples
©2002 John Wiley & Sons, Inc. Tradução autorizada da edição em língua inglesa publicada pela John Wiley & Sons, Inc.
All rights reserved.
ISBN 0-471-38365-1

Capa: *Mário Röhnelt*

Leitura Final: *Carla Krohn*

Supervisão editorial: *Arysinha Jacques Affonso e Denise Weber Nowaczyk*

Editoração eletrônica: *Laser House*

Java é uma marca da Sun Microsystems, Inc.

Unix® é uma marca registrada nos Estados Unidos e em outros países e licenciada pela X/Open Company, Ltd.

Todos os demais nomes dos produtos mencionados nesta obra são marcas registradas em nome dos respectivos proprietários.

Reservados todos os direitos de publicação, em língua portuguesa, à
ARTMED® EDITORA S.A.
(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S.A.)
Av. Jerônimo de Ornelas, 670 - Santana
90040-340 Porto Alegre RS
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

SÃO PAULO
Av. Angélica, 1091 - Higienópolis
01227-100 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Sobre os autores

Os professores Goodrich e Tamassia são conhecidos pesquisadores de estruturas de dados e algoritmos, tendo publicado vários artigos, nesta área, sobre aplicações na Internet, visualização de informação, sistemas de informação geográfica, segurança de computadores e projeto auxiliado por computador. Eles têm um longo histórico de colaboração e foram pesquisadores-chefes em vários projetos financiados pela National Science Foundation, pelo Army Research Office e pela Defense Advanced Research Projects Agency. Atuam também em pesquisa em tecnologia educacional com ênfase especial em sistemas para visualização de algoritmos e infra-estrutura para ensino à distância.

Michael Goodrich recebeu seu Ph. D. em Ciência da Computação pela Universidade de Purdue em 1987. É professor no Departamento de Informação e Ciência da Computação da University of California, Irvine. Antes disso, foi professor de Ciência da Computação da Johns Hopkins University e diretor do Hopkins Center for Algorithm Engineering. É também editor do *International Journal of Computational Geometry & Applications*, do *Journal of Computational and Systems Sciences* e do *Journal of Graph Algorithms and Applications*.

Roberto Tamassia recebeu seu Ph. D. em Engenharia Elétrica e de Computação pela Universidade de Illinois em Urbana-Champaign, em 1988. Atualmente, é professor no Departamento de Ciência da Computação e diretor do Centro de Geometria Computacional da Brown University. Também é editor das revistas *Computational Geometry: Theory and Applications* e *Journal of Graph Algorithms and Applications*, já tendo sido do corpo editorial da *IEEE Transactions on Computers*.

Além de seus trabalhos em pesquisa, os autores também têm grande experiência em ensino. Por exemplo, o Dr. Goodrich ensina estruturas de dados e algoritmos na Johns Hopkins University desde 1987, em cursos básicos e avançados. Recebeu vários prêmios nessa função. Em sala de aula, costuma envolver os alunos em animados debates que despertam a intuição sobre as técnicas envolvendo estruturas de dados e algoritmos, bem como a formulação de soluções cuja análise é matematicamente rigorosa. O Dr. Tamassia ensina estruturas de dados e algoritmos em cursos introdutórios na Brown University desde 1988. Ele também atraiu muitos estudantes para a Geometria Computacional, que é uma disciplina de pós-graduação com muitos alunos na Brown. Seu estilo de ensinar distingue-se pelo uso de apresentações multimídia interativas, seguindo a tradição da "sala de aula eletrônica" da Brown. As páginas Web cuidadosamente projetadas dos cursos ministrados pelo Dr. Tamassia são usadas como referência por estudantes e profissionais de todo o mundo.

Para meus filhos Paul, Anna e Jack.
– *Michael T. Goodrich*

Para Isabel
– *Roberto Tamassia*

Agradecimentos

Várias pessoas nos auxiliaram com os conteúdos deste livro. Em especial agradecemos a Jeff Achter, Ryan Baker, David Borland, Ulrik Brades, Stina Bridgemann, Robert Cohen, David Emory, David Ginat, Natasha Gelfand, Mark Handy, Benoît Hudson, Jeremy Mullendore, Daniel Polivy, John Schultz, Andrew Schewerin, Michael Shin, Galina Shubina e Lucca Vismara.

Somos gratos a todos os nossos assistentes de ensino, que ajudaram a desenvolver exercícios, tarefas de programação e sistemas de animação de algoritmos. Houve um grande número de amigos e colegas cujos comentários conduziram a melhorias no texto. Somos particularmente gratos a Karen Goodrich, Art Moorshead e Scott Smith por suas valiosas sugestões. Somos também gratos aos leitores e revisores externos por seus muitos comentários, *e-mails* e críticas construtivas, extremamente úteis.

Agradecemos também a nossos editores, Paul Crocket e Bill Zobrist, por seu apoio entusiástico durante este projeto. A equipe de produção da Wiley foi ótima. Muito obrigado às pessoas que nos ajudaram com o desenvolvimento do livro, incluindo Susannah Barr, Katherine Hepburn, Bonnie Kubat, Sharon Prendergast, Marc Ranger, Jeri Warner e Jennifer Welter.

Este manuscrito foi preparado originalmente com LaTeX para o texto e Adobe Framemaker® e Visio® para as figuras. O sistema LGrind foi usado para formatar os trechos de código Java no LaTeX. O sistema de controle de versão permitiu uma coordenação tranqüila dos nossos (algumas vezes concorrentes) arquivos de edição.

Finalmente, gostaríamos de agradecer a Isabel Cruz, Karen Goodrich, Guiseppe di Battista, Franco Preparata, Ioannis Tollis e a nossos pais por prover conselho, encorajamento e apoio nos vários estágios da preparação deste livro. Também agradecemos a eles por nos lembrarem que existem outras coisas na vida além de escrever livros.

Prefácio

Este livro foi concebido para oferecer uma introdução abrangente ao projeto e à análise de algoritmos e às estruturas de dados. Considerando os currículos de ciência da computação e engenharia de computação, escrevemos o texto com o objetivo de focar, basicamente, um curso avançado de algoritmos, o que corresponde a alunos no primeiro ano de graduação em algumas faculdades.

Tópicos

Os assuntos cobertos foram selecionados a partir de um conjunto amplo de tópicos de projeto e análise de algoritmos, incluindo os seguintes:

- **Projeto e análise de algoritmos**, incluindo notação assintótica, análise de pior caso, amortização, randomização e análise experimental.
- **Algoritmos para padrões de projeto**, incluindo o método guloso, divisão e conquista, programação dinâmica, *backtracking* e *branch-and-bound*.
- ***Frameworks* algorítmicos**, incluindo NP-completude, algoritmos de aproximação, algoritmos *on-line*, algoritmos de memória externa, algoritmos distribuídos e algoritmos paralelos.
- **Estruturas de dados**, incluindo listas, vetores árvores, filas de prioridades, árvores AVL, árvores 2-4, *splay-trees*, árvores-B, tabelas *hash*, *skip lists*, árvores *union-find*.
- **Algoritmos combinatórios**, incluindo *heapsort*, *quick-sort*, *merge-sort*, seleção, lista de colocações paralela e ordenação em paralelo.
- **Algoritmos de grafos**, incluindo caminhamentos (DFS e BFS), ordenação topológica, caminho mínimo (todos os vértices e um vértice), árvores de cobertura mínima, fluxo máximo, fluxo de custo mínimo e associações.
- **Algoritmos geométricos**, incluindo pesquisa intervalar, cobertura convexa, interseção de segmentos e pares mais próximos.
- **Algoritmos numéricos**, incluindo multiplicação de inteiros, matrizes e polinômios, Transformada Rápida de Fourier (FFT*), algoritmo de Euclides estendido, exponenciação modular e teste primal.
- **Algoritmos para Internet**, incluindo roteamento de pacotes, *multicasting*, eleição do líder, encriptação, assinaturas digitais, detecção de padrões de texto, recuperação de informações, compressão de dados, *cache* para Web e leilões para Web.

* N. de R. Em inglês, *Fast Fourier Transform*.

Para o professor

Este livro foi concebido como livro-texto para alunos de um curso avançado de algoritmos, que pode ser um curso para alunos de primeiro ano da graduação em algumas faculdades. Este livro contém muitos exercícios, divididos em exercícios de reforço, exercícios de criatividade e projetos de implementação. Alguns aspectos deste livro foram especialmente projetados tendo o professor em mente, incluindo:

- **Justificativas visuais (isto é, a figura prova)**, o que torna os argumentos matemáticos mais compreensíveis para os estudantes, atraindo os que têm melhor compreensão visual. Um exemplo de justificativa visual é nossa análise da construção *bottom-up* de um *heap*. O tópico tradicionalmente é de difícil entendimento para os estudantes, exigindo, desta forma, muito tempo dos instrutores nas explicações. A prova visual incluída é intuitiva, rigorosa e rápida.
- **Padrões de projeto de algoritmos**, que oferecem técnicas genéricas de projeto e implementação de algoritmos. Exemplos incluem divisão e conquista, programação dinâmica, o padrão *decorator* e o padrão do método padrão.
- **Uso de randomização**, que tira vantagem das escolhas aleatórias em algoritmos para simplificar seu projeto e análise. Este uso substitui análises complexas de caso médio de estruturas de dados sofisticadas pela análise de simples estruturas de dados e algoritmos. Exemplos incluem listas *skip*, *quick-sort* randômico, *quick-select* randômico e teste primal randomizado.
- **Tópicos sobre algoritmos para Internet**, que tanto motivam tópicos tradicionais de algoritmos sob uma nova perspectiva como apresentam novos algoritmos derivados de aplicações para Internet. Exemplos incluem recuperação de informações, busca na Web, roteamento de pacotes, algoritmos de leilão para Web e algoritmos de busca na Web. Acreditamos que motivar o estudo de algoritmos com aplicações para Internet melhora significativamente o interesse do aluno pelo estudo de algoritmos.
- **Exemplos de implementação Java**, que cobrem métodos de projeto de *software*, aspectos de implementação orientada a objetos e análise experimental de algoritmos. Estes exemplos de implementação, oferecidos em seções separadas de diversos capítulos, são opcionais, de maneira que o instrutor pode tratar dos mesmos em suas aulas, indicá-los como leitura adicional ou simplesmente deixá-los de lado.

O livro também está estruturado de maneira a permitir ao professor grande liberdade na forma de organizar e apresentar o material. Desta forma, a dependência entre os capítulos é flexível, permitindo ao instrutor montar um curso de algoritmos destacando os tópicos que considera mais importantes. Discutimos tópicos relativos a algoritmos para Internet de maneira extensiva, pois provaram ser interessantes para os alunos. Além disso, incluímos em diversos pontos exemplos de aplicação na Web de algoritmos tradicionais.

A Tabela 0.1 apresenta o modo como este livro pode ser usado em um curso tradicional de introdução a algoritmos, ainda que com alguns tópicos novos motivados pela Internet.

Cap.	Tópico	Opção
1	Análise de algoritmos	Análise experimental
2	Estruturas de dados	Exemplo de um *heap* em Java
3	Pesquisa	Incluir um dos § 3.2-3.5
4	Ordenação	*In-place quick-sort*
5	Técnicas algorítmicas	A TRF
6	Algoritmos de grafos	Exemplo de DFS em Java
7	Grafos balanceados	Exemplo do algoritmo de Dijkstra em Java
8	Combinações e fluxo	Incluir no final do curso
9	Processamento de texto (pelo menos uma seção)	Experimentos
12	Geometria computacional	Incluir no final do curso
13	NP-completude	*Backtracking*
14	*Frameworks* (pelo menos um)	Incluir no final do curso

Tabela 0.1 Exemplo de seqüência para plano de ensino de um curso tradicional de Introdução a Algoritmos, incluindo escolhas opcionais para cada capítulo.

Este livro pode também ser usado para um curso especializado em algoritmos para Internet, que revisa alguns tópicos tradicionais usando uma motivação baseada na Internet, cobrindo também novos algoritmos que são derivados de aplicações Internet. Mostramos na Tabela 0.2 como este livro pode ser usado em tal curso.

Cap.	Tópico	Opção
1	Análise de algoritmos	Análise experimental
2	Estruturas de dados (incluindo *hashing*)	Revisão rápida
3	Pesquisa (incluindo § 3.5, listas *skip*)	Exemplo de árvore de pesquisa em Java
4	Ordenação	*In-place quick-sort*
5	Técnicas algorítmicas	A TRF
6	Algoritmos de grafos	Exemplo de DFS em Java
7	Grafos balanceados	Pular algum dos algoritmos MST
8	Combinações e fluxo	Algoritmos de combinação
9	Processamento de texto (pelo menos uma seção)	Identificação de padrões
10	Segurança & criptografia	Exemplos Java
13	NP-completude	Incluir no final do curso
14	*Frameworks* (pelo menos dois)	Incluir no final do curso

Tabela 0.2 Exemplo de seqüência para plano de ensino de um curso de Algoritmos para Internet, incluindo escolhas opcionais para cada capítulo.

Naturalmente, outras opções são possíveis, incuindo um misto de um curso tradicional de algoritmos e um curso de algoritmos para Internet. Não repassaremos o assunto, deixando a criatividade na organização para o professor interessado.

Tópicos na Web

Este livro vem acompanhado de um grande *Website**:

http://www.wiley.com/college/goodrich/

Este *site* inclui material auxiliar que amplia os tópicos deste livro. Especificamente para estudantes, incluímos:

- apostilas das apresentações (em formato de quatro *slides* por página) para a maioria dos tópicos do livro;
- um banco de dados de dicas em exercícios selecionados, indexados pelo número do problema;
- *applets* interativos que animam as estruturas de dados e algoritmos tradicionais;
- código-fonte para os exemplos em Java deste livro.

Pressentimos que o servidor de dicas pode despertar um interesse especial, particularmente no que se refere aos problemas de criatividade que podem ser um tanto desafiadores para alguns estudantes.

Os professores que usarem este livro podem contar uma parte do *website* dedicada apenas a eles, incluindo os seguintes tópicos de auxilio:

- solução para exercícios selecionados do livro;
- um banco de dados com exercícios adicionais e suas soluções;
- apresentações (em formato de um *slide* por página) para a maioria dos tópicos cobertos por este livro.

Os leitores interessados na implementação dos algoritmos e estruturas de dados podem baixar a JSDL, a biblioteca de estruturas de dados em Java, de

http://www.jdsl.org

Pré-requisitos

Escrevemos presumindo que o leitor já tenha algum conhecimento do tema. Em particular supomos que tenha uma compreensão básica das estruturas de dados elementares, tais como arranjos e listas encadeadas, e esteja pelo menos vagamente familiarizado com uma linguagem de programação de alto nível, tal como C, C++ ou Java. Mesmo assim, todos os algoritmos são descritos usando um "pseudocódigo" de alto nível, e construções específicas de linguagens de programação são usadas apenas nas seções opcionais de exemplos Java.

Em termos de fundamentação matemática, supomos que o leitor esteja familiarizado com os tópicos de matemática do primeiro ano de faculdade, incluindo exponentes, logaritmos, somatórios, limites e probabilidade elementar. Mesmo assim, revisamos a maioria desses itens no Capítulo 1, incluindo expoentes, logaritmos e somatórios, e apresentamos um resumo de outros itens matemáticos úteis, incluindo probabilidade elementar, no Apêndice.

* N. de R. O conteúdo do *site* está em inglês.

Sumário

Parte I Ferramentas de análise

1. Análise de algoritmos 19

1.1 Metodologias para análise de algoritmos 21
1.2 Notação assintótica 28
1.3 Uma rápida revisão matemática. 35
1.4 Exemplos de análise assintótica de algoritmos 45
1.5 Amortização 48
1.6 Experimentação 55
1.7 Exercícios 59

2. Estruturas de dados básicas 68

2.1 Pilhas e filas 70
2.2 Vetores, listas e seqüências 77
2.3 Árvores 86
2.4 Filas de prioridades e *heaps* 104
2.5 Dicionários e tabelas *hash* 122
2.6 Exemplo em Java: *heap* 135
2.7 Exercícios 138

3. Árvores de pesquisa e *skip lists* 147

3.1 Dicionários ordenados e árvores binárias 149
3.2 Árvores AVL 159
3.3 Árvores de pesquisa com profundidade limitada. 165
3.4 *Splay trees* 187
3.5 *Skip lists* 198
3.6 Exemplo em Java: árvores AVL e árvores vermelho-pretas 205
3.7 Exercícios 215

4. Ordenação, conjuntos e seleção 221

4.1 *Merge-sort* 223
4.2 O TAD conjunto. 229
4.3 *Quick-sort* 239
4.4 Limite inferior em ordenação baseada em comparação 243
4.5 *Bucket-sort* e *radix-sort* 245
4.6 Comparação dos algoritmos de ordenação. 248
4.7 Seleção 249
4.8 Exemplo em Java: *in-place quick-sort* 252
4.9 Exercícios 255

5. Técnicas fundamentais 262

5.1 O método guloso 264
5.2 Divisão e conquista 267
5.3 Programação dinâmica 277
5.4 Exercícios 285

Parte II Algoritmos para grafos

6. Algoritmos para grafos 291
6.1 O tipo abstrato de dados grafo 293
6.2 Estruturas de dados para grafos 299
6.3 Caminhamento de grafos 305
6.4 Grafos dirigidos. 319
6.5 Exemplo em Java: caminhamento em profundidade. 331
6.6 Exercícios 337

7. Grafos ponderados 342
7.1 Caminhos mínimos a partir de um vértice 344
7.2 Caminhos mínimos entre todos os vértices. 356
7.3 Árvores de cobertura mínima. 361
7.4 Exemplo em Java: o algoritmo de Dijkstra 372
7.5 Exercícios 376

8. Fluxo em redes e associação 380
8.1 Fluxos e cortes 382
8.2 Fluxo máximo 386
8.3 A associação máxima bipartida 395
8.4 Fluxo de custo mínimo 397
8.5 Exemplo em Java: fluxo de custo mínimo 403
8.6 Exercícios 409

Parte III Algoritmos para Internet

9. Algoritmos para Internet 415
9.1 Cadeias de caracteres e pesquisa de padrões 417
9.2 *Tries* 426
9.3 Compressão de textos 435
9.4 Teste de similaridade de textos 438
9.5 Exercícios 442

10. Teoria dos números e criptografia 447
10.1 Algoritmos fundamentais envolvendo números. 449
10.2 Processos criptográficos 465
10.3 Algoritmos e protocolos para segurança de informações 474
10.4 A transformada rápida de Fourier 481
10.5 Exemplo em Java: a FFT 492
10.6 Exercícios 500

11. Algoritmos de redes 504
11.1 Modelos e medidas de complexidade 506
11.2 Algoritmos distribuídos fundamentais 510
11.3 Roteamento *broadcast* e *unicast* 522
11.4 Roteamento *multicast*. 526
11.5 Exercícios 532

Parte IV Tópicos adicionais

12. Geometria computacional 539
12.1 Árvores intervalares 540
12.2 Árvores de pesquisa por prioridade 547
12.3 *Quadtrees* e árvores *k*-D 552
12.4 A técnica do plano rotacional 556
12.5 Cobertura convexa 562
12.6 Exemplo em Java: cobertura convexa 573
12.7 Exercícios 577

13. NP-completude 582
13.1 **P** e **NP** 584
13.2 **NP**-completude 590
13.3 Problemas **NP**-completos importantes 593
13.4 Agoritmos aproximativos 608
13.5 *Backtracking* e *branch-and-bound* 616
13.6 Exercícios 627

14. *Frameworks* algorítmicos 633
14.1 Algoritmos de memória externa 635
14.2 Algoritmos paralelos 646
14.3 Algoritmos *on-line* 655
14.4 Exercícios 668

A Fatos matemáticos úteis 673

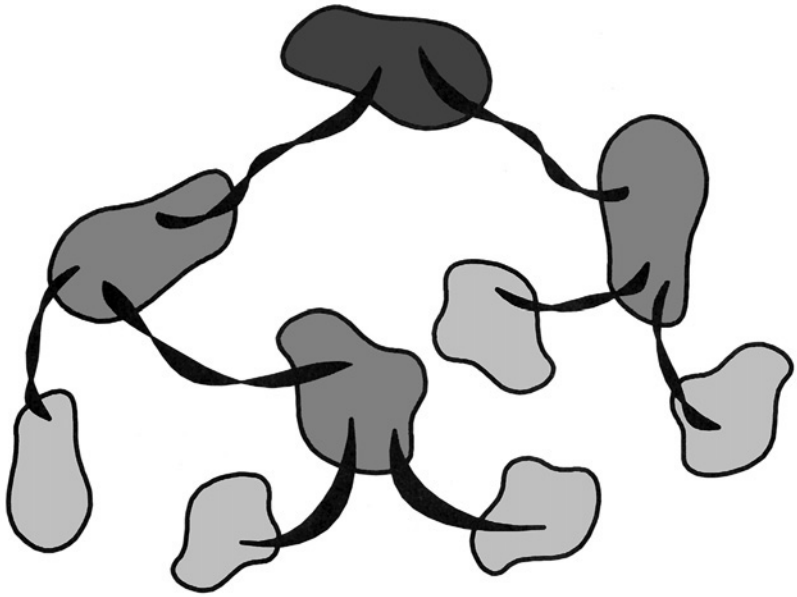
Bibliografia 677

Índice 687

Parte

Ferramentas de análise

I



Capítulo

Análise de algoritmos

1

1.1 Metodologias para análise de algoritmos 21
1.1.1 Pseudocódigo 22
1.1.2 O modelo da Máquina de Acesso Aleatório (RAM) 24
1.1.3 Contagem de operações primitivas. 25
1.1.4 Análise de algoritmos recursivos 27
1.2 Notação assintótica 28
1.2.1 Notação *O* 28
1.2.2 Parentes de *O* 31
1.2.3 Importância da análise assintótica 33
1.3 Uma rápida revisão matemática 35
1.3.1 Somatórios 36
1.3.2 Logaritmos e expoentes 37
1.3.3 Técnicas simples de justificativa 39
1.3.4 Probabilidade básica. 42
1.4 Exemplos de análise assintótica de algoritmos 45
1.4.1 Um algoritmo de tempo quadrático 45
1.4.2 Um algoritmo de tempo linear 47
1.5 Amortização 48
1.5.1 Técnicas de amortização 49
1.5.2 Análise de uma implementação de arranjo expansível 52
1.6 Experimentação 55
1.6.1 Escolha do experimento 55
1.6.2 Análise de dados e visualização. 57
1.7 Exercícios 59

Em uma história clássica, foi pedido ao famoso matemático Arquimedes que determinasse se uma coroa dourada encomendada pelo rei era realmente de ouro puro e não continha prata, como um informante havia afirmado. Arquimedes descobriu uma maneira de resolver o problema quando estava entrando na banheira. Ele notou que a água transbordava da banheira à medida que entrava e, percebendo as implicações desse fato, ele imediatamente saiu do banho e correu nu pelas ruas da cidade gritando "*Heureka*, *heureka*!", pois tinha descoberto uma ferramenta de análise (o deslocamento) que, quando combinada com uma escala, poderia determinar se a nova coroa do rei era realmente boa ou não. Essa foi uma descoberta infeliz para o ourives, pois, quando Arquimedes fez sua análise, a coroa deslocou mais água que o mesmo peso em ouro puro, mostrando que a coroa não era, de fato, ouro puro.

Neste livro, estamos interessados no projeto de "boas" estruturas de dados e algoritmos. De forma simples, uma ***estrutura de dados*** é uma maneira sistemática de organizar e acessar dados, e um ***algoritmo*** é um procedimento passo a passo para realizar alguma tarefa em um tempo finito. Esses conceitos são centrais para a computação, mas para sermos capazes de classificar estruturas de dados e algoritmos como "bons", devemos ter maneiras precisas de analisá-los.

A ferramenta básica de análise que usaremos neste livro envolve a caracterização do tempo de execução de algoritmos e operações sobre estruturas de dados, com seu consumo de memória também sendo de interesse. O tempo de execução é uma medida natural, pois tempo é um recurso precioso, mas o foco no tempo de execução como medida básica implica que teremos de usar um pouco de matemática para descrever os tempos de execução e analisar algoritmos.

Começaremos este capítulo descrevendo a metodologia básica para analisar algoritmos, que inclui a linguagem para descrição de algoritmos, o modelo computacional para o qual a linguagem foi planejada e os fatores principais que consideramos quando analisamos o tempo de execução. Também incluímos uma breve descrição da forma como algoritmos recursivos são analisados. Na Seção 1.2, apresentamos a notação principal usada para caracterizar tempos de execução – a chamada "Notação O". Essas são as principais ferramentas teóricas para projetar e analisar algoritmos.

Na Seção 1.3, interrompemos nosso desenvolvimento da metodologia básica de análise de algoritmos para revisar alguns fatos matemáticos importantes, incluindo discussões sobre somatórios, logaritmos, técnicas de prova e probabilidade básica. Com esse conhecimento e a notação para análise de algoritmos, apresentamos alguns estudos em análise teórica de algoritmos na Seção 1.4. Seguimos esses exemplos na Seção 1.5, apresentando uma interessante técnica de análise chamada de amortização, que nos permite contabilizar o comportamento conjunto de uma série de operações individuais. Finalmente, na Seção 1.6, concluímos o capítulo discutindo uma importante e prática técnica de análise – a experimentação. Discutiremos os princípios de uma boa metodologia experimental bem como técnicas para resumir e caracterizar dados de uma análise experimental.

1.1 Metodologias para análise de algoritmos

O tempo de execução de um algoritmo ou operação sobre uma estrutura de dados depende tipicamente de uma série de fatores. Então, qual seria a maneira mais adequada de medi-lo? Se um algoritmo foi implementado, podemos estudar o tempo gasto por ele executando-o com vários dados de entrada e registrando o tempo gasto em cada execução. Essas medições podem ser feitas de maneira acurada usando-se chamadas de sistema que são incluídas na linguagem ou sistema operacional no qual o algoritmo foi implementado. Em geral, estamos interessados em determinar a dependência do tempo de execução com respeito ao tamanho da entrada fornecida ao algoritmo. Para determiná-la, podemos realizar vários experimentos com diferentes entradas de vários tamanhos. Poderemos então visualizar os resultados desses experimentos plotando o desempenho de cada execução do algoritmo como um ponto com coordenada x igual ao tamanho n da entrada fornecida e coordenada y igual ao tempo de execução t (veja a Figura 1.1). Para ser significativa, essa análise exige que escolhamos bons exemplos de entradas e que sejam feitos testes suficientes para que possam ser feitas afirmações válidas sob um ponto de vista estatístico, uma abordagem que discutiremos na Seção 1.6.

Em geral, o tempo de execução de um algoritmo ou de um método de uma estrutura de dados cresce com o tamanho da entrada, embora possa variar para entradas diferentes porém de mesmo tamanho. Adicionalmente, o tempo de execução também é afetado pelo *hardware* onde o algoritmo é executado (processador, frequência, memória, disco, etc.) e pelo *software* (sistema operacional, linguagem de programação, compilador, interpretador, etc.) sobre o qual o algoritmo é implementado, compilado e executado. Se todos os outros fatores forem iguais, o tempo de execução do mesmo algoritmo para os mesmos dados de entrada será menor se o computador tiver, por exemplo, um processador muito mais rápido ou se a implementação for feita como um programa escrito em código de máquina em vez de uma implementação interpretada em uma máquina virtual.

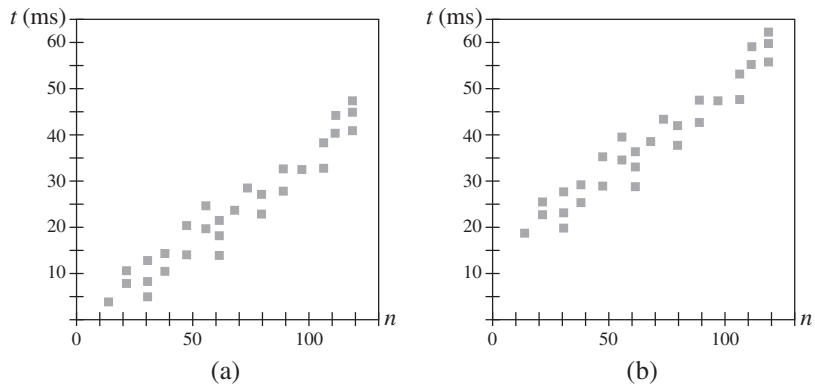


Figura 1.1 Resultados de um estudo experimental sobre o tempo de execução de um algoritmo. Um ponto com coordenadas (n,t) indica que, para uma entrada de tamanho n, o algoritmo teve um tempo de execução de t milissegundos (ms). (a) O algoritmo foi executado em um computador veloz; (b) o algoritmo foi executado em um computador mais lento.

Requisitos para uma metodologia geral de análise

Estudos experimentais dos tempos de execução são úteis, como exploraremos na Seção 1.6, mas eles tem algumas limitações:

- Experimentos podem ser feitos apenas em um número limitado de entradas de teste e deve-se tomar cuidado para assegurar que elas sejam representativas.
- É difícil comparar a eficiência de dois algoritmos a não ser que os experimentos para a obtenção de seus tempos de execução tenham sido feitos com o mesmo *hardware* e *software*.
- É necessário implementar e executar um algoritmo para poder estudar seu tempo de execução experimentalmente.

Assim, enquanto a experimentação tem um papel importante na análise de algoritmos, ela sozinha não é suficiente. Portanto, além da experimentação, desejamos uma metodologia analítica que:

- leve em conta todas as entradas possíveis;
- permita-nos avaliar a eficiência relativa de dois algoritmos de uma forma independente do *hardware* e do *software* onde eles são implementados;
- possa ser feita através do estudo de uma descrição em alto nível do algoritmo sem que seja necessário implementá-lo ou executá-lo.

Esta metodologia objetiva associar a cada algoritmo uma função $f(n)$ que caracteriza o tempo de execução do algoritmo como uma função do tamanho n da entrada. Funções típicas que serão encontradas incluem n e n^2. Por exemplo, escreveremos afirmações como "O algoritmo A é executado em tempo proporcional a n", querendo dizer que, se fossemos realizar experimentos, constataríamos que o tempo de execução do algoritmo A para ***qualquer*** entrada de tamanho n nunca excede cn, onde c é uma constante que depende do *hardware* e do *software* usados no experimento. Dados dois algoritmos A e B, onde A é executado em tempo proporcional a n e B em tempo proporcional a n^2, preferiremos o algoritmo A ao algoritmo B, uma vez que a função n cresce mais devagar do que a função n^2.

Estamos prontos para arregaçar as mangas e começar a desenvolver a metodologia para análise de algoritmos. Existem vários componentes para esta metodologia, incluindo os seguintes:

- uma linguagem para descrição de algoritmos;
- um modelo computacional para execução de algoritmos;
- uma métrica para medir o tempo de execução de algoritmos;
- uma abordagem para caracterizar os tempos de execução, incluindo aqueles de algoritmos recursivos.

1.1.1 Pseudocódigo

Programadores são freqüentemente solicitados a descrever algoritmos de uma forma que seja adequada especialmente a outros seres humanos. Essas descrições não são programas de computador, mas são mais estruturados do que uma simples prosa. Tais descrições também facilitam a análise de alto nível de uma estrutura de dados ou algoritmos. Essas descrições são chamadas de **pseudocódigo**.

Um exemplo de pseudocódigo

Dado um arranjo A composto de n elementos inteiros, suponha que desejamos encontrar o maior elemento. Para resolver este problema, podemos usar um algoritmo chamado arrayMax, que percorre os elementos de A usando um laço **para**.

A descrição em pseudocódigo do algoritmo arrayMax é mostrada no Algoritmo 1.2.

Algoritmo arrayMax(A,n):
  Entrada: um arranjo A com $n \geq 1$ elementos inteiros.
  Saída: o maior elemento em A.
 currentMax ← $A[0]$
 para i ← 1 **até** $n - 1$ **faça**
  se *currentMax* < $A[i]$ **então**
   currentMax ← $A[i]$
 retorne *currentMax*

Algoritmo 1.2 O algoritmo arrayMax.

Note que o pseudocódigo é mais compacto do que o código equivalente em uma linguagem de programação. Adicionalmente o pseudocódigo é mais claro para ser lido e compreendido.

Uso do pseudocódigo para provar a correção do algoritmo

Inspecionando o pseudocódigo, podemos argumentar sobre a correção do algoritmo arrayMax de forma simples. A variável *currentMax* inicia sendo igual ao primeiro elemento de A. Afirmamos que no começo da i-ésima iteração do laço, a variável *currentMax* é igual ao maior dos primeiros i elementos em A. Como comparamos *currentMax* com $A[i]$ na iteração i, se esta afirmação for verdadeira antes desta iteração, será verdadeira depois dela para $i + 1$ (que é o próximo valor do contador i). Assim, após $n - 1$ iterações, a variável *currentMax* será igual ao maior elemento de A. Assim como neste exemplo, desejamos que nossas descrições em pseudocódigo sejam sempre detalhadas o bastante para justificar a correção do algoritmo descrito e simples o bastante para que um leitor humano as compreenda.

O que é pseudocódigo?

Pseudocódigo é uma mistura de linguagem natural e estruturas de programação de alto nível usada para descrever as idéias principais da implementação genérica de uma estrutura de dados ou algoritmo. No entanto, não existe uma definição precisa da linguagem de ***pseudocódigo*** por causa de seu uso da linguagem natural. Ao mesmo tempo, para aumentar sua clareza, o pseudocódigo mistura linguagem natural com construções-padrão de linguagens de programação. As construções de linguagens de programação que escolhemos são coerentes com linguagens modernas de alto nível como C, C++ e Java. Essas construções incluem as seguintes:

- ***Expressões:*** usamos símbolos matemáticos para escrever expressões numéricas e booleanas. Usamos uma seta para a esquerda (←) para representar o operador de atribuição em comandos de atribuição (equivalente ao operador = em C, C++ e Java) e usamos o sinal de igual (=) como a relação de igualdade em expressões booleanas (de forma equivalente ao operador == em C, C++ e Java).

- ***Declarações de métodos:*** **Algoritmo** nome(*param*1, *param*2,...) declara um novo método chamado "nome" e declara seus parâmetros.
- ***Estruturas de decisão:*** **se** condição **então** ações-verdade [**também** ações-falsidade]. Usamos indentação para indicar quais ações são incluídas nas ações-verdade e ações-falsidade.
- ***Laços enquanto:*** **enquanto** condição **faça** ações. Usamos indentação para indicar quais ações são incluídas no laço.
- ***Laços repita:*** **repita** ações **até** condição. Usamos indentação para indicar quais ações são incluídas no laço.
- ***Laços para*****: para** definição de variável e incremento **faça** ações. Usamos indentação para indicar quais ações são incluídas no laço.
- ***Indexação de arranjos:*** $A[i]$ representa a i-ésima posição do arranjo A. As posições de um arranjo A de n posições variam de $A[0]$ a $A[n-1]$ (coerentemente com Java).
- ***Chamadas de métodos:*** objeto.método(args) (objeto é opcional se o contexto for claro).
- ***Retorno de métodos:*** **retorne** valor. Esta operação retorna o valor especificado ao método chamador. Quando escrevemos pseudocódigo, devemos ter em mente que o estamos fazendo para outras pessoas, e não para um computador. Assim, devemos nos esforçar para comunicar idéias de alto nível, e não detalhes menores de implementação. Ao mesmo tempo, não devemos desprezar passos importantes. Como muitas formas de comunicação humana, encontrar o equilíbrio adequado é uma habilidade importante refinada com a prática.

Agora que desenvolvemos uma forma de alto nível para descrever algoritmos, vamos discutir como caracterizar analiticamente algoritmos escritos em pseudocódigo.

1.1.2 O modelo da Máquina de Acesso Aleatório (RAM)

Como vimos anteriormente, a análise experimental é importante, mas tem suas limitações. Se desejamos analisar um algoritmo em particular sem realizar experimentos para medir seu tempo de execução, podemos usar a abordagem a seguir diretamente sobre o código de alto nível ou pseudocódigo. Definimos um conjunto de ***operações primitivas*** de alto nível independentes da linguagem de programação usada e que podem ser identificadas no pseudocódigo. As seguintes operações estão incluídas entre as operações primitivas:

- atribuição de valores a variáveis;
- chamadas de métodos;
- operações aritméticas (por exemplo, adição de dois números);
- comparação de dois números;
- acesso a um arranjo;
- seguir uma referência a um objeto;
- retorno de um método.

Mais especificamente, uma operação primitiva corresponde a uma instrução de baixo nível com um tempo de execução constante, mas que depende do ambiente de *softwa-*

re e *hardware*. Em vez de tentar determinar o tempo de execução específico de cada operação primitiva, vamos simplesmente ***contar*** quantas operações primitivas serão executadas e usar este número t como uma estimativa de alto nível do tempo de execução do algoritmo. Essa contagem de operações está relacionada com o tempo de execução em um *hardware* e um *software* específicos, pois cada operação corresponde a uma instrução realizada em tempo constante e existe um número fixo de operações primitivas. Nesta abordagem, presume-se implicitamente que os tempos de execução de operações primitivas diferentes serão similares. Assim, o número t de operações primitivas que um algoritmo realiza será proporcional ao tempo de execução daquele algoritmo.

Definição do modelo RAM

Essa abordagem de simplesmente contar o número de operações primitivas dá origem a um modelo computacional chamado ***Máquina de Acesso Aleatório*** (RAM – *random access machine*). Este modelo, que não deve ser confundido com "memória de acesso randômico", vê um computador como uma unidade central de processamento (ou CPU) conectada a uma memória. Cada posição da memória armazena uma palavra, que pode ser um número, uma cadeia de caracteres ou um endereço, ou seja, algum tipo básico da linguagem. O termo "acesso randômico" refere-se à capacidade da CPU de acessar uma posição arbitrária de memória em apenas uma operação primitiva. Para manter o modelo simples, não colocamos limites no tamanho dos números que podem ser armazenados nas posições de memória. Presumimos que a CPU do modelo RAM pode realizar qualquer operação primitiva em um número constante de passos que não depende do tamanho da entrada. Assim, um limite acurado para o número de operações primitivas que um algoritmo realiza corresponde diretamente ao tempo de execução daquele algoritmo sob o modelo RAM.

1.1.3 Contagem de operações primitivas

Mostraremos agora como contar o número de operações primitivas executadas por um algoritmo, usando como exemplo o algoritmo arrayMax, cujo pseudocódigo e implementação em Java foram mostrados no Algoritmo 1.2. Fazemos essa análise concentrando-nos em cada passo do algoritmo e contando o número de operações primitivas que ele leva, considerando que algumas operações são repetidas, pois estão dentro de um laço.

- A inicialização da variável *currentMax* com o valor de $A[0]$ corresponde a duas operações primitivas (indexar um arranjo e atribuir um valor a uma variável) e é executada apenas uma vez quando o algoritmo é iniciado. Assim, ela contribui em duas unidades para a contagem.
- No início do laço **para**, o contador i é inicializado com o valor 1. Essa ação corresponde a uma operação primitiva (atribuir um valor a uma variável).
- Antes de entrar no laço **para**, a condição $i < n$ é verificada. Essa ação corresponde a uma operação primitiva (comparar dois números). Como o contador i é inicializado em 0 e incrementado em 1 no fim de cada iteração do laço, a comparação $i < n$ é feita n vezes. Assim, a condição contribui com n unidades para a contagem.

- O corpo do laço **para** é executado $n - 1$ vezes (para valores 1, 2,..., $n - 1$ do contador). A cada iteração, $A[i]$ é comparado com *currentMax* (duas operações primitivas, indexação e comparação), o valor de $A[i]$ pode ser atribuído a *currentMax* (duas operações primitivas, indexação e atribuição) e o contador i é incrementado (duas operações, soma e atribuição). Assim, a cada iteração do laço, quatro ou seis operações primitivas são realizadas dependendo se $A[i] \leq currentMax$ ou se $A[i] > currentMax$. Desta forma, o corpo do laço contribui para a contagem com um número de unidades que varia entre $4(n - 1)$ e $6(n - 1)$.
- Retornar o valor da variável *currentMax* corresponde a uma operação primitiva e é executada apenas uma vez.

Resumindo, o número de operações primitivas $t(n)$ executadas pelo algoritmo arrayMax é no mínimo

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

e no máximo

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2.$$

O melhor caso ($t(n) = 5n$) ocorre quando $A[0]$ é o maior elemento e, portanto, a variável *currentMax* não tem seu valor alterado posteriormente. O pior caso ($t(n) = 7n - 2$) ocorre quando o arranjo A tem seus elementos em ordem crescente, de forma que *currentMax* é alterada a cada iteração.

Análise de caso médio e de pior caso

Como no caso do método arrayMax, um algoritmo pode ser mais rápido sobre algumas entradas do que sobre outras. Nesses casos, podemos desejar expressar o tempo de execução deste algoritmo como uma média calculada com todas as entradas possíveis. Infelizmente, embora valioso, este tipo de ***caso médio*** é bastante desafiador. Ele requer que seja definida uma distribuição de probabilidades sobre o conjunto das entradas, o que é geralmente uma tarefa difícil. A Figura 1.3 mostra esquematicamente como, de acordo com a distribuição das entradas, o tempo de execução de um algoritmo pode ter qualquer valor entre o tempo para o pior caso e o tempo para o melhor caso. Por exemplo, o que acontecerá se as entradas forem apenas dos tipos "A" e "D"?

Uma análise de caso médio tipicamente requer que sejam calculados os tempos de execução baseados em uma distribuição de probabilidade. Esse tipo de análise freqüentemente exige matemática sofisticada e teoria das probabilidades.

Portanto, exceto para estudos experimentais ou a análise de algoritmos que são randomizados, no restante deste livro caracterizaremos os tempos de execução em termos do ***pior caso***. Diremos, portanto, que o algoritmo arrayMax executa $t(n) = 7n - 2$ ***no pior caso***, significando que o maior número de operações primitivas executadas pelo algoritmo, em quaisquer entradas possíveis de tamanho n, é $7n - 2$.

Esse tipo de análise é muito mais fácil do que uma análise de caso médio, já que não requer teoria das probabilidades. De fato, requer apenas a habilidade de identificar a pior entrada possível, o que geralmente é simples. Além disso, usar uma abordagem de pior caso pode conduzir a algoritmos melhores: tendo certeza de que um algoritmo tem bom desempenho no pior caso garante que ele tem bom desempenho em ***todos*** os casos. Ou seja, projetar para o pior caso pode levar a algoritmos mais "fortes", da mesma forma que um atleta pode fortalecer seus músculos correndo morro acima.

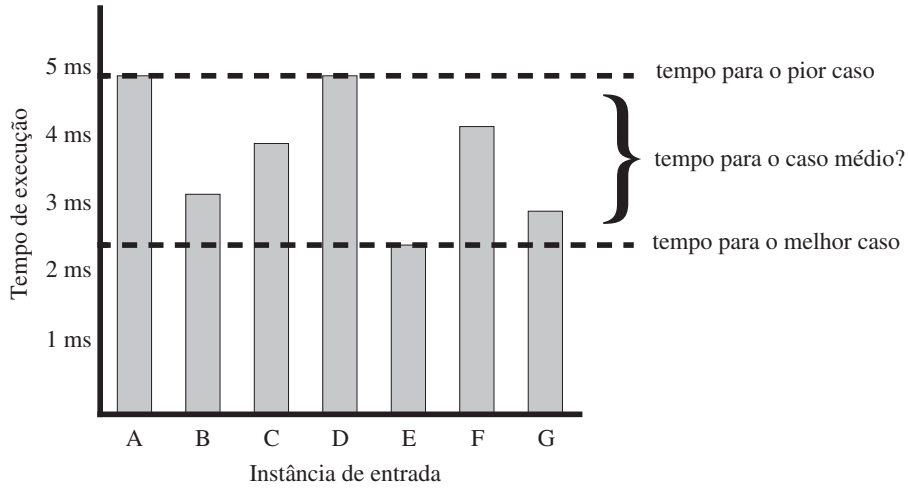


Figura 1.3 A diferença entre tempo para o pior caso e o tempo para o melhor caso. Cada barra representa o tempo de execução de um algoritmo sobre uma entrada diferente.

1.1.4 Análise de algoritmos recursivos

Iterar não é a única forma interessante de resolver problemas. Outra técnica útil, usada por muitos algoritmos, é o uso da ***recursão***. Nesta técnica, definimos um procedimento P que pode usar a si mesmo como uma sub-rotina desde que as chamadas a P sejam feitas para resolver um problema de tamanho menor. As chamadas a P para instâncias menores do problema são chamadas de "chamadas recursivas". Um procedimento recursivo deve sempre definir um ***caso base***, que é pequeno o bastante para que o algoritmo o resolva diretamente sem usar recursão.

Oferecemos uma solução recursiva para o problema do máximo elemento de um arranjo no Algoritmo 1.4. Esse algoritmo primeiramente verifica se o arranjo contém um único item, que será o máximo neste caso. Assim, neste caso base, podemos resolver o problema imediatamente; senão, o algoritmo recursivamente determina o máximo dos primeiros $n - 1$ elementos no arranjo e retorna o máximo entre estes valores e o último elemento do arranjo.

Como neste exemplo, algoritmos recursivos são freqüentemente bastante elegantes. No entanto, analisar o tempo de execução de um algoritmo recursivo custa um pouco de trabalho adicional. Em particular, para analisar o tempo de execução, usamos uma ***equação de recorrência***, que define restrições matemáticas que o tempo de execução de um algoritmo recursivo deve seguir. Introduzimos uma função $T(n)$ que denota o tempo de execução do algoritmo com uma entrada de tamanho n e escrevemos equações que $T(n)$ deve satisfazer. Por exemplo, caracterizamos o tempo de execução $T(n)$ do algoritmo recursiveMax como

$$T(n) = \begin{cases} 3 & \text{se } n = 1 \\ T(n-1) + 7 & \text{do contrário,} \end{cases}$$

presumindo que contamos cada comparação, acesso a arranjo, chamada recursiva, avaliação de maior elemento ou comando **retorne** como uma operação primitiva. Ideal-

mente gostaríamos de caracterizar uma equação de recorrência como a anterior em ***forma fechada***, em que não há referências à função T no lado direito. Para o algoritmo recursiveMax, não é difícil verificar que uma forma fechada seria $T(n) = 7(n - 1) + 3 = 7n - 4$. Em geral, determinar soluções em forma fechada para equações de recorrência pode ser muito mais desafiador do que isto, e estudaremos alguns exemplos de equações de recorrência no Capítulo 4, quando estudaremos algoritmos de classificação e de seleção. Estudaremos métodos para resolver equações de recorrência de uma forma geral na Seção 5.2.

Algoritmo recursiveMax(A, n):
  Entrada: um arranjo A com $n \geq 1$ elementos inteiros.
  Saída: o maior elemento em A.
  se $n = 1$ **então**
    retorne $A[0]$
  retorne max {recursiveMax(A, $n - 1$), $A[n - 1]$}

Algoritmo 1.4 O algoritmo recursiveMax.

1.2 Notação assintótica

Tivemos de entrar em um elevado grau de detalhamento para analisar o tempo de execução de um algoritmo simples como arrayMax e seu primo recursivo recursiveMax. Essa análise seria bastante trabalhosa se fosse aplicada para algoritmos mais complicados. Em geral, cada passo em uma descrição em pseudocódigo e cada comando em uma implementação em linguagem de alto nível corresponde a um pequeno número de operações primitivas que não depende do tamanho da entrada. Assim, podemos realizar uma análise simplificada que estima o número de operações primitivas executadas, exceto por um fator constante, simplesmente contando os passos do pseudocódigo ou os comandos da linguagem de alto nível executada. Felizmente, existe uma notação que nos permite caracterizar os fatores principais que afetam o tempo de execução de um algoritmo sem entrar em todos os detalhes sobre exatamente quantas operações primitivas são realizadas para cada conjunto de instruções de tempo constante.

1.2.1 Notação *O*

Sejam $f(n)$ e $g(n)$ funções mapeando inteiros não negativos em números reais. Dizemos que $f(n)$ é $O(g(n))$ se existe uma constante real $c > 0$ e uma constante inteira $n_0 \geq 1$, tais que $f(n) \leq cg(n)$ para todo inteiro $n \geq n_0$. Essa definição é geralmente chamada de notação O, pois geralmente se diz "$f(n)$ é ***O*** de $g(n)$", ou também "$f(n)$ é ***ordem*** $g(n)$". Esta definição é ilustrada na Figura 1.5.

Exemplo 1.1: $7n - 2$ é $O(n)$.

Prova: *pela definição da notação O, precisamos achar uma constante* $c > 0$ *e uma constante inteira* $n_0 > 1$*, tais que* $7n - 2 \leq cn$ *para todo inteiro* $n \geq n_0$*. É fácil perceber que uma escolha poderia ser* $c = 7$ *e* $n_0 = 1$*. De fato, esta é uma das infinitas escolhas possíveis, porque qualquer número real maior ou igual a* 7 *será uma escolha possível para c, e qualquer inteiro maior ou igual a* 1 *é uma escolha possível para* n_0*.* ■

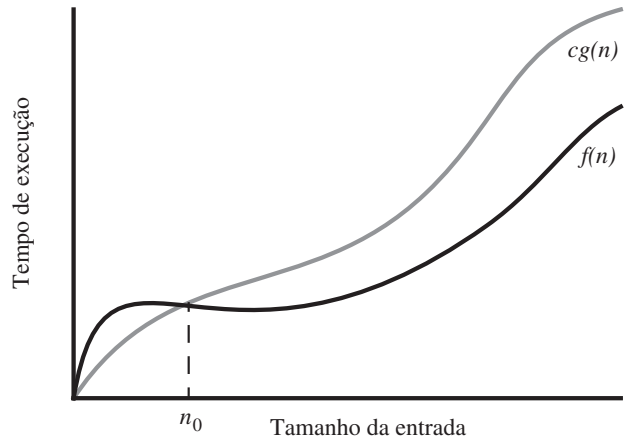


Figura 1.5 Ilustração da notação O. A função $f(n)$ é $O(g(n))$, pois $f(n) \leq c \cdot g(n)$ quando $n \geq n_0$.

A notação O permite-nos dizer que uma função de n é "menor ou igual a" outra função (pela desigualdade "$\leq$ " na definição), descontando-se um fator constante (a constante c na definição) e ***assintoticamente*** à medida que n cresce para infinito (pela condição de que "$n \geq n_0$" na definição).

A notação O é muito usada para caracterizar o tempo de execução e consumo de memória em função de um parâmetro n, que varia de problema para problema, mas é geralmente definido como uma noção intuitiva do tamanho do mesmo. Por exemplo, se estamos interessados em achar o maior elemento em um arranjo de inteiros (veja o algoritmo arrayMax, dado no Algoritmo 1.2), seria natural fazer n representar o número de elementos no arranjo. Por exemplo, podemos escrever a seguinte (e matematicamente precisa) afirmação sobre o tempo de execução do algoritmo arrayMax do Algoritmo 1.2:

Teorema 1.2: *o tempo de execução do algoritmo* arrayMax *para determinar o maior elemento de um arranjo de n inteiros é O(n).*

Prova: como mostrado na Seção 1.1.3, o número de operações primitivas executadas pelo algoritmo é no máximo $7n - 2$. Portanto, aplicamos a definição de O com $c = 7$ e $n_0 = 1$ e concluímos que o tempo de execução do algoritmo arrayMax é $O(n)$. ■

Vamos considerar mais alguns exemplos que ilustram a notação O.

Exemplo 1.3: $20n^3 + 10n\log(n) + 5$ é $O(n^3)$.

Prova: $20n^3 + 10n\log(n) + 5 \leq 35n^3$, para $n \geq 1$. ■

De fato, qualquer polinômio $a_k n^k + a_{k-1}n^{k-1} + ... + a_0$ é $O(n^k)$.

Exemplo 1.4: $3\log(n) + \log(\log(n))$ é $O(\log(n))$.

Prova: $3\log(n) + \log(\log(n)) \leq 4\log(n)$ *para* $n \geq 2$. *Note que* $\log(\log(n))$ *não é definida para* $n = 1$, *e por isso usamos* $n = 2$. ■

Exemplo 1.5: 2^{100} *é* $O(1)$.

Prova: $2^{100} \leq 2^{100} \cdot 1$, *para* $n \geq 1$. *Note que a variável n não aparece na desigualdade, já que estamos lidando com funções de valor constante.* ■

Exemplo 1.6: $5/n$ *é* $O(1/n)$.

Prova: $5/n \leq 5(1/n)$ *para* $n \geq 1$ (*mesmo que esta seja na verdade uma função* **decrescente**). ■

Em geral, devemos usar a notação O para caracterizar uma função tão detalhadamente quanto possível. Mesmo sendo verdade que a função $f(n) = 4n^3 + 3n^{4/3}$ é $O(n^5)$ ou mesmo $O(n^3\log(n))$, é mais correto dizer que $f(n)$ é $O(n^3)$. Considere a seguinte analogia: um viajante com fome dirige por uma estrada do interior e passa por um fazendeiro que está indo para casa. Quando o viajante pergunta quanto ele ainda tem de dirigir até achar um restaurante, o fazendeiro responde que "com certeza são menos de 12 horas". Isto pode estar correto, mas é muito mais informativo (e preciso) se ele disser "você encontra um mercado em mais cinco ou dez minutos".

Em vez de sempre aplicar a definição da notação O diretamente para obter uma estimativa, podemos usar as seguintes regras para simplificar a notação.

Teorema 1.7: *Sejam* $d(n)$, $e(n)$, $f(n)$ *e* $g(n)$ *funções mapeando inteiros não negativos em reais não negativos. Então*

1. Se $d(n)$ é $O(f(n))$, então $ad(n)$ é $O(f(n))$ para qualquer constante $a > 0$.
2. Se $d(n)$ é $O(f(n))$ e $e(n)$ é $O(g(n))$, então $d(n) + e(n)$ é $O(f(n) + g(n))$.
3. Se $d(n)$ é $O(f(n))$ e $e(n)$ é $O(g(n))$, então $d(n)e(n)$ é $O(f(n)g(n))$.
4. Se $d(n)$ é $O(f(n))$ e $f(n)$ é $O(g(n))$, então $d(n)$ é $O(g(n))$.
5. Se $f(n)$ é um polinômio de grau d (ou seja, $f(n) = a_0 + a_1n + ... + a_dn^d$), então $f(n)$ é $O(n^d)$.
6. n^x é $O(a^n)$ para quaisquer constantes $x > 0$ e $a > 1$.
7. $\log(n^x)$ é $O(\log(n))$ para qualquer constante $x > 0$.
8. $(\log(n))^x$ é $O(n^y)$ para quaisquer constantes $x > 0$ e $y > 0$.

Considera-se inadequado incluir fatores constantes e termos de menor ordem na notação O. Por exemplo, não é bem-aceito dizer que a função $2n^2$ é $O(4n^2 + 6n\log(n))$, embora isso seja correto. Devemos nos esforçar para descrever a função na notação O em seus termos ***mais simples***.

Exemplo 1.8: $2n^3 + 4n^2\log(n)$ é $O(n^3)$.

Prova: aplicamos as regras do Teorema 1.7 na seguinte ordem:

- $\log(n)$ é $O(n)$ (Regra 8).
- $4n^2\log(n)$ é $O(4n^3)$ (Regra 3).
- $2n^3 + 4n^2\log(n)$ é $O(2n^3 + 4n^3)$ (Regra 2).
- $2n^3 + 4n^3$ é $O(n^3)$ (Regra 1 ou Regra 5).
- $2n^3 + 4n^2\log(n)$ é $O(n^3)$ (Regra 4). ■

Algumas funções aparecem freqüentemente na análise de algoritmos e estruturas de dados e, seguidamente usamos termos especiais para nos referir a elas. O termo **linear** é usado com freqüência, por exemplo, para se referir à classe de funções que são $O(n)$. A Tabela 1.6 descreve esse e outros termos comumente usados na análise de algoritmos.

logarítmica	linear	quadrática	polinomial	exponencial
$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^k)$ com $k \geq 1$	$O(a^n)$ com $a > 1$

Tabela 1.6 Terminologia de classes de funções.

Uso da notação *O*

É considerado pouco elegante dizer "$f(n) \leq O(g(n))$" já que a notação O por si mesma transmite a idéia de "menor ou igual". Da mesma forma, embora comum, não é correto escrever "$f(n) = O(g(n))$" (com a relação de " = " mantendo seu sentido usual) e é errado escrever "$f(n) \geq O(g(n))$" ou "$f(n) > O(g(n))$". O mais apropriado é dizer que "$f(n)$ **é** $O(g(n))$". Para o leitor com maior inclinação matemática, também é correto dizer

$$\text{“}f(n) \in O(g(n)),\text{”}$$

pois a notação O denota, tecnicamente, um conjunto de funções.

Mesmo sob essa interpretação, existe considerável liberdade para usarmos operações aritméticas com a notação O, desde que a ligação com a definição da notação seja clara. Por exemplo, podemos dizer

$$\text{“}f(n) \text{ é } g(n) + O(h(n)),\text{”}$$

o que significaria que existem constantes $c > 0$ e $n_0 \geq 1$ para as quais $f(n) \leq g(n) + ch(n)$ para $n \geq n_0$. De fato, podemos às vezes querer mostrar precisamente o termo mais importante em uma caracterização assintótica e, neste caso, diríamos que "$f(n)$ é $g(n) + O(h(n))$", onde $h(n)$ cresce mais devagar do que $g(n)$. Por exemplo, podemos dizer que $2n\log(n) + 4n + 10\sqrt{n}$ é $2n\log(n) + O(n)$.

1.2.2 Parentes de *O*

Assim como a notação O fornece uma maneira assintótica de dizer que uma função é "menor ou igual a" outra função, as notações seguintes fornecem maneiras assintóticas de fazer outros tipos de comparações.

Ômega e Teta

Sejam $f(n)$ e $g(n)$ funções mapeando números inteiros em números reais. Dizemos que $f(n)$ é $\Omega(g(n))$ (dito "$f(n)$ é ômega de $g(n)$") se $g(n)$ é $O(f(n))$; ou seja, existe uma constante $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \geq cg(n)$ para $n \geq n_0$. Essa definição permite-nos dizer que uma função é assintoticamente maior que ou igual a outra, exceto por um fator constante. Da mesma forma, dizemos que $f(n)$ é $\Theta(g(n))$ (dito "$f(n)$ é teta de $g(n)$") se $f(n)$ é $O(g(n))$ e $f(n)$ é $\Omega(g(n))$; ou seja, existem constantes $c' > 0$ e $c'' > 0$ e uma constante inteira $n_0 \geq 1$ tais que $c'g(n) \leq f(n) \leq c''g(n)$ para $n \geq n_0$.

A definição de *Teta* permite-nos dizer que duas funções são assintoticamente iguais, exceto por um fator constante. Consideraremos a seguir alguns exemplos dessa notação.

Exemplo 1.9: $3\log(n) + \log(\log(n))$ é $\Omega(\log(n))$.

Prova: $3\log(n) + \log(\log(n)) \geq 3\log(n))$ para $n \geq 2$. ■

Esse exemplo mostra que os termos de menor ordem não são dominantes quando estabelecemos limites inferiores com a notação Ω. Da mesma forma, como mostra o próximo exemplo, esses termos também não são dominantes na notação Θ.

Exemplo 1.10: $3\log(n) + \log(\log(n))$ é $\Theta(\log(n))$.

Prova: resulta dos Exemplos 1.4 e 1.9. ■

Palavras de cautela

Algumas palavras de cautela sobre a notação assintótica podem ser dadas neste ponto. Primeiramente, note que o uso da notação O e suas parentes pode ser um pouco confuso se os fatores constantes que elas escondem forem muito altos. Por exemplo, enquanto é verdade que a função $10^{100}n$ é $\Theta(n)$, se este for o tempo de execução de um algoritmo sendo comparado a um outro cujo tempo de execução é $10n \log(n)$, preferiremos o algoritmo com tempo de execução $\Theta(10n \log(n))$, mesmo que o primeiro algoritmo seja linear e, portanto assintoticamente mais rápido. Esta preferência se justifica pelo fator constante, 10^{100}, que é chamado de "um *googol*". Muitos astrônomos acreditam que esse número seja um limite para o número de átomos no universo observável e, por isso, é improvável que tenhamos algum problema do mundo real com esse tamanho de entrada. Assim, mesmo usando a notação O, devemos estar conscientes dos fatores constantes e dos termos de mais baixa ordem que estão "escondidos".

A observação anterior conduz à discussão do que seja um algoritmo "rápido". De forma geral, qualquer algoritmo rodando em tempo $O(n \log(n))$ (e com um fator constante razoável) pode ser considerado eficiente. Mesmo um algoritmo que seja $O(n^2)$ pode ser suficientemente rápido em alguns contextos, ou seja, quando n é pequeno. Por outro lado, um algoritmo rodando em tempo $\Theta(2^n)$ não pode nunca ser considerado eficiente.

Esse fato é ilustrado com uma história famosa sobre o inventor do jogo de xadrez. Ele pediu que seu rei lhe pagasse um grão de arroz pela primeira casa do tabuleiro, dois pela segunda, quatro pela terceira, oito pela quarta e assim por diante. Tente imaginar 2^{64} grãos de arroz empilhados na última casa! Este número nem mesmo pode ser representado como um inteiro longo na maioria das linguagens de programação.

Se devemos traçar uma linha entre algoritmos eficientes e ineficientes, portanto, é natural fazer esta distinção entre os algoritmos que rodam em tempo polinomial e aqueles que requerem tempo exponencial. Ou seja, fazemos a distinção entre algoritmos que rodam em tempo $O(n^k)$ para alguma constante $k \geq 1$ e aqueles cujo tempo de execução é $\Theta(c^n)$ para alguma constante $c > 1$. Assim como outras noções discutidas nesta seção, esta também deve ser acolhida com um grão de sal, ou seja, com cautela, pois um algoritmo rodando em tempo $\Theta(n^{100})$ com certeza não deve ser considerado muito eficiente. Mesmo assim, a distinção entre algoritmos de tempo polinomial e algoritmos de tempo exponencial é considerada uma medida robusta de tratabilidade.

"Primos distantes" de O: o e ω

Existem maneiras de dizer que uma função é estritamente menor ou estritamente maior (assintoticamente) do que outra função, mas estas formas não são tão usadas quanto as notações O, Ω e Θ. Mesmo assim, para uma descrição completa, damos aqui suas definições.

Sejam $f(n)$ e $g(n)$ funções mapeando números inteiros em números reais. Dizemos que $f(n)$ é $o(g(n))$ (dito "$f(n)$ é 'o' pequeno de $g(n)$" se para qualquer constante $c > 0$ existe uma constante inteira $n_0 > 0$ tal que $f(n) \leq cg(n)$ para $n > n_0$.

Da mesma forma, dizemos que $f(n)$ é $\omega(g(n))$ (dito "$f(n)$ é ômega pequeno de $g(n)$") se $g(n)$ é $o(f(n))$, ou seja, se para qualquer constante $c > 0$ existe uma constante inteira $n_0 > 0$ tal que $g(n) < cf(n)$ para $n \geq n_0$. Intuitivamente, $o(.)$ é assintoticamente análogo a "menor que", e $\omega(.)$ é análogo a "maior que" em um sentido assintótico.

Exemplo 1.11: a função $f(n) = 12n^2 + 6n$ é $o(n^3)$ e $\omega(n)$.

Prova: *primeiramente vamos mostrar que $f(n)$ é $o(n^3)$. Seja $c > 0$ uma constante. Se tomarmos $n_0 = (12 + 6)/c$, então para $n \geq n_0$ teremos*

$$cn^3 \geq 12n^2 + 6n^2 \geq 12n^2 + 6n.$$

Portanto, $f(n)$ é $o(n^3)$.

Para mostrar que $f(n)$ é $\omega(n)$, usamos uma constante $c > 0$. Se $n_0 = c/12$, então para $n > n_0$ teremos

$$12n^2 + 6n \geq 12n^2 + cn. \qquad \blacksquare$$

Portanto, $f(n)$ é $\omega(n)$.

Para o leitor familiarizado com limites, notamos que $f(n)$ é $o(g(n))$ se e somente se

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$

se este limite existir. A diferença principal entre o e O é que $f(n)$ é $O(g(n))$ se ***existem*** constantes $c > 0$ e $n_0 \geq 1$ tais que $f(n) \leq cg(n)$ para $n \geq n_0$, enquanto $f(n)$ é $o(g(n))$ se ***para todas*** as constantes $c > 0$ existe n_0 tal que $f(n) \leq cg(n)$ para $n \geq n_0$. Intuitivamente, $f(n)$ é $o(g(n))$ se $f(n)$ se torna insignificante se comparado a $g(n)$ à medida que n cresce infinitamente. Como dito anteriormente, a notação assintótica é útil porque permite que nos concentremos no fator principal controlando o crescimento de uma função.

Resumindo, as notações assintóticas O, Ω e Θ bem como o e ω fornecem uma linguagem conveniente para a análise de estruturas de dados e algoritmos. Como mencionado anteriormente, essas noções são convenientes porque permitem que nos concentremos nos aspectos gerais do comportamento de um algoritmo, em vez de seus detalhes.

1.2.3 Importância da análise assintótica

A notação assintótica traz vários benefícios importantes que não são imediatamente óbvios. Especificamente, ilustramos um importante aspecto do ponto de vista assintótico na Tabela 1.7. Essa tabela explora o maior tamanho permitido para os dados de entrada de um problema para os vários tempos de execução, se o problema deve ser re-

solvido em 1 segundo, 1 minuto e 1 hora, presumindo que cada operação é processada em 1 microssegundo ($1\mu s$). Ela também mostra a importância do bom projeto de um algoritmo, pois um algoritmo assintoticamente demorado (por exemplo, um que seja $O(n^2)$) é superado por um algoritmo com tempo assintoticamente mais rápido (por exemplo, um que seja $O(n\log(n))$), mesmo que o fator constante do algoritmo assintoticamente mais rápido seja maior.

Tempo de execução	Tamanho (n) máximo de problemas 1 segundo	1 minuto	1 hora
$400n$	2.500	150.000	9.000.000
$20n\lceil\log n\rceil$	4.096	166.666	7.826.087
$2n^2$	707	5.477	42.426
n^4	31	88	244
2^n	19	25	31

Tabela 1.7 Tamanho máximo de problemas que podem ser resolvidos em um segundo, um minuto e uma hora para vários tempos de execução medidos em microssegundos.

A importância do bom projeto de algoritmos vai além do que pode ser resolvido eficientemente em um dado computador. Como mostrado na Tabela 1.8, mesmo se o *hardware* for acelerado dramaticamente, ainda assim não poderemos superar o problema representado por um algoritmo assintoticamente lento. A tabela mostra o novo tamanho máximo de problema que pode ser resolvido (em qualquer tempo dado), presumindo que os algoritmos como tempo de execução dado são executados em um computador 256 vezes mais rápido do que o anterior.

Tempo de execução	Novo tamanho máximo de problemas
$400n$	$256m$
$20n\lceil\log n\rceil$	aprox. $256((\log m)/(7+\log m))m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m+8$

Tabela 1.8 Aumento no tamanho máximo do problema que pode ser resolvido em um dado tempo usando-se um computador 256 vezes mais rápido do que o anterior, para diversos tempos de execução de algoritmos. Cada entrada é dada em função de m, o tamanho máximo de problema dado anteriormente.

Ordenando funções por suas taxas de crescimento

Suponha que dois algoritmos podem resolver o mesmo problema: um algoritmo A que tem um tempo de execução $\Theta(n)$ e um algoritmo B com tempo de execução $\Theta(n^2)$. Qual deles é melhor? A notação o diz que n é $o(n^2)$, e isso implica que o algoritmo A é

assintoticamente melhor do que o algoritmo B, embora para algum dado valor (pequeno) de n seja possível que B tenha um tempo de execução menor do que o algoritmo A.

Podemos usar a notação o para ordenar classes de funções por seu crescimento assintótico. A lista de funções na Tabela 1.9 está ordenada por ordem de crescimento e uma função $f(n)$ precede uma função g(n) se $f(n)$ é $o(g(n))$.

Funções ordenadas por ordem de crescimento
$\log n$
$\log^2 n$
$\sqrt{n}$
n
$n \log n$
n^2
n^3
2^n

Tabela 1.9 Uma lista ordenada de funções simples. Note que, usando-se a terminologia comum, uma das funções acima é logarítmica, duas são polilogarítmicas, três são sublineares, uma é linear, uma é quadrática, uma é cúbica e uma é exponencial.

Na Tabela 1.10, ilustramos a diferença na taxa de crescimento de todas exceto uma das funções da Tabela 1.9.

n	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	2^n
2	1	1,4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2,8	8	24	64	512	256
16	4	4	16	64	256	4.096	65.536
32	5	5,7	32	160	1.024	32.768	4.294.967.296
64	6	8	64	384	4.096	262.144	$1{,}84 \times 10^{19}$
128	7	11	128	896	16.384	2.097.152	$3{,}40 \times 10^{38}$
256	8	16	256	2.048	65.536	16.777.216	$1{,}15 \times 10^{77}$
512	9	23	512	4.608	262.144	134.217.728	$1{,}34 \times 10^{154}$
1.024	10	32	1.024	10.240	1.048.576	1.073.741.824	$1{,}79 \times 10^{308}$

Tabela 1.10 Crescimento de diversas funções.

1.3 Uma rápida revisão matemática

Nesta seção, revisaremos alguns conceitos fundamentais de matemática discreta que irão surgir em várias de nossas discussões. Além destes conceitos fundamentais, incluímos no Apêndice A uma lista de outros fatos matemáticos úteis que são aplicados na análise de estruturas de dados e análise de algoritmos.

1.3.1 Somatórios

Uma notação que aparece com freqüência na análise de algoritmos e de estruturas de dados é o ***somatório***, definido a seguir:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b).$$

Somatórios surgem na análise de algoritmos e estruturas de dados porque o tempo de execução de laços pode ser representado naturalmente com somas. Por exemplo, uma soma que surge freqüentemente na análise de algoritmos e estruturas de dados é a progressão geométrica.

Teorema 1.12: *para qualquer inteiro* $n \geq 0$ *e qualquer número real* $0 < a \neq 1$, *considere*

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$

(*lembrando que* $a^0 = 1$ *se* $a > 0$). *Esta soma é igual a*

$$\frac{1 - a^{n+1}}{1 - a}.$$

Somatórios como os do Teorema 1.12 são chamados de somas ***geométricas***, porque cada termo é geometricamente maior do que o anterior se $a > 1$. Ou seja, os termos em uma soma geométrica exibem crescimento exponencial. Por exemplo, qualquer pessoa trabalhando em computação deveria saber que

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

pois este é o maior inteiro que pode ser representado em notação binária usando *n bits*.

Outra soma que surge em vários contextos é

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

Essa soma surge na análise de laços em casos em que o número de operações efetuadas dentro do laço aumenta em um valor fixo a cada iteração. Essa soma também tem uma história interessante. Em 1787, um professor alemão de escola primária decidiu manter seus alunos de 9 e 10 anos de idade ocupados com a tarefa de somar todos os números de 1 a 100. Logo depois de apresentar o exercício, uma das crianças afirmou ter a resposta. O professor ficou desconfiado, pois o aluno tinha apenas a resposta em suas anotações, sem nenhum cálculo. Mas a resposta estava correta: 5050. Aquele estudante era Carl Friedrich Gauss, que cresceria para ser um dos maiores matemáticos do século XIX. Acredita-se que o jovem Gauss obteve a resposta para a tarefa usando a identidade a seguir.

Teorema 1.13: *para qualquer inteiro* $n \geq 1$, *temos*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Prova: damos duas justificativas visuais do Teorema 1.13 na Figura 1.11, ambas baseadas na área de uma coleção de retângulos representando os números de 1 a n. Na Figura 1.11a, desenhamos um grande triângulo sobre os retângulos, notando que a área dos retângulos é igual à área do triângulo ($n^2/2$) mais a área de n pequenos triângulos, cada um de área 1/2. Na Figura 1.11b, que vale apenas para quando n é par, notamos que 1 mais n é $n + 1$, bem como 2 mais $n - 1$, 3 mais $n - 2$ e assim por diante. Temos $n/2$ desses pares. ■

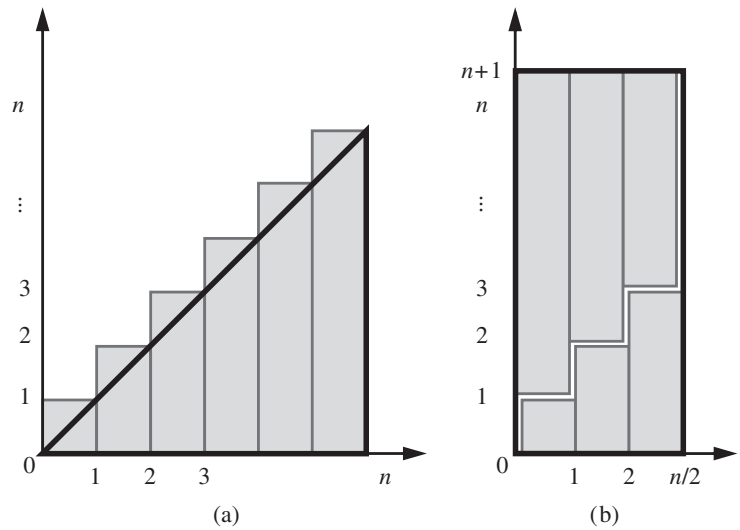


Figura 1.11 Justificativas visuais do Teorema 1.13. Ambas as ilustrações apresentam a identidade em termos da área total coberta por n retângulos de largura unitária com alturas 1,2, ..., n. Em (a) os retângulos cobrem um grande triângulo de área $n^2/2$ (base n e altura n) mais n pequenos triângulos de área 1/2 cada (base 1 e altura 1). Em (b), que vale apenas para n par, os retângulos cobrem um grande retângulo de base $n/2$ e altura $n + 1$.

1.3.2 Logaritmos e expoentes

Um dos interessantes e até surpreendentes aspectos da análise de estruturas de dados e algoritmos é a freqüente presença de logaritmos e expoentes, onde dizemos

$$\log_b a = c \quad \text{se} \quad a = b^c.$$

Como é costume na literatura de computação, omitiremos escrever a base b dos logaritmos quando $b = 2$. Por exemplo, log (1024) = 10.

Existem algumas regras importantes para logaritmos e expoentes, incluindo as seguintes:

Teorema 1.14: *sejam a, b e c números positivos reais. Temos:*

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_c a) / \log_c b$

5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a / b^c = b^{a-c}$

Como uma notação abreviada, usamos $\log^c(n)$ denotando a função $(\log(n))^c$ e usamos $\log\log(n)$ para denotar a função $\log(\log(n))$. Em vez de mostrar como poderíamos derivar cada uma das identidades anteriores a partir das definições de logaritmos e expoentes, vamos ilustrar essas identidades com alguns exemplos de suas utilidades.

Exemplo 1.15: *ilustramos alguns casos interessantes quando a base do logaritmo é 2. As regras citadas se referem ao Teorema 1.14.*

- $\log(2n \log n) = 1 + \log n + \log\log n$, *pela regra 1 (duas vezes)*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *pela regra 2*
- $\log\sqrt{n} = \log(n)^{1/2} = (\log n)/2$, *pela regra 3*
- $\log\log\sqrt{n} = \log(\log n)/2 = \log\log n - 1$, *pelas regras 2 e 3*
- $\log_4 n = (\log n) / \log 4 = (\log n)/2$, *pela regra 4*
- $\log 2^n = n$, *pela regra 3*
- $2^{\log n} = n$, *pela regra 5*
- $2^{2\log n} = (2^{\log n})^2 = n^2$, *pelas regras 5 e 6*
- $4^n = (2^2)^n = 2^{2n}$, *pela regra 6*
- $n^2\, 2^{3\log n} = n^2 \cdot n^3 = n^5$, *pelas regras 5, 6 e 7*
- $4^n / 2^n = 2^{2n}/2^n = 2^{2n-n} = 2^n$, *pelas regras 6 e 8*

As funções piso e teto

É adequado fazer um comentário adicional sobre logaritmos. O valor de um logaritmo tipicamente não é inteiro, ainda que o tempo de execução de um algoritmo seja usualmente expresso por um número inteiro, como por exemplo o número de operações efetuadas. Assim, a análise de um algoritmo pode às vezes envolver o uso das funções conhecidas como "piso" e "teto", que são definidas respectivamente como segue:

- $\lfloor x \rfloor$ = o maior inteiro menor ou igual a x.
- $\lceil x \rceil$ = o menor inteiro maior ou igual a x.

Essas funções fornecem uma maneira de converter funções com valores reais em funções com valores inteiros. Mesmo assim, funções usadas para analisar estruturas de dados e algoritmos são freqüentemente expressas como funções reais (por exemplo, $n \log(n)$ ou $n^{3/2}$). Devemos interpretar um tempo de execução como se ele tivesse uma função teto o circundando.[1]

[1] Funções de tempo de execução com valores reais quase sempre são usadas juntamente com a notação assintótica descrita na Seção 1.2, para a qual o uso de funções "teto" seria redundante (veja o Exercício R-1.24).

1.3.3 Técnicas simples de justificativa

Algumas vezes, desejaremos fazer afirmações sobre uma determinada estrutura de dados ou algoritmo. Podemos, por exemplo, querer mostrar que nosso algoritmo é correto ou que é rápido. Para fazer tais afirmações de forma rigorosa, devemos usar uma argumentação matemática, justificando ou ***provando*** nossas afirmações. Felizmente, existem várias maneiras simples de fazer isso.

Através de exemplos

Algumas afirmações tem a forma genérica "existe um elemento x no conjunto S que tem a propriedade P". Para justificar tal afirmação, precisamos apenas encontrar um x em S que tenha a propriedade P. Outras afirmações são da forma "todo elemento x no conjunto S tem a propriedade P". Para mostrar que essa afirmação é falsa, precisamos apenas mostrar um x do conjunto S que não tenha a propriedade P. Um tal x é chamado de ***contra-exemplo***.

Exemplo 1.16: *um certo Professor Entrenós afirma que todo número da forma* $2^i - 1$ *é primo, se i é maior do que* 1. *O professor está errado.*

Prova: *para provar que o Professor Entrenós está errado, precisamos achar um contra-exemplo. Felizmente não precisamos procurar muito, pois* $2^4 - 1 = 16 - 1 = 15 = 3 \cdot 5$. ■

O ataque "contra"

Outro conjunto de técnicas envolve o uso de negações. Os dois métodos básicos são o uso do ***contrapositivo*** e da ***contradição***. O uso do contrapositivo é como olhar em um espelho negativo: para justificar a afirmação "se p é verdade então q é verdade", estabelecemos a verdade da afirmação "se q não é verdade então p não é verdade". Logicamente essas duas afirmações são equivalentes, mas a segunda, que é a ***contrapositiva*** da primeira, pode ser mais fácil de demonstrar.

Exemplo 1.17: *Se ab é ímpar, então a é ímpar ou b é par.*

Prova: *para justificar esta afirmação, considere seu contrapositivo "se a é par e b é ímpar, então ab é par". Assim, suponha* $a = 2i$ *para algum inteiro i. Neste caso,* $ab = (2i)b = 2(ib)$. *Portanto, ab é par.* ■

Além de mostrar o uso do contrapositivo, o exemplo anterior também contém uma aplicação da ***Lei de De Morgan***. Essa lei ajuda-nos a lidar com negações, pois ela afirma que a negação de "p ou q" assume a forma "não p e não q".

Outra técnica de justificativa por negação envolve o uso da ***contradição***, que freqüentemente também envolve o uso da Lei de De Morgan. Aplicando a técnica, estabelecemos que uma afirmação q é verdadeira supondo primeiro que ela é falsa e mostrando que esta suposição leva a uma contradição (tal como $2 \neq 2$ ou $1 > 3$). Chegando a uma contradição, estaremos mostrando que, se q for falsa, teremos uma situação inconsistente, e portanto q deve ser verdadeira. Naturalmente, para chegar a essa conclusão, devemos ter certeza de que temos uma situação consistente ainda antes de supor que q é falsa.

Exemplo 1.18: *se ab é ímpar, então a é ímpar ou b é par.*

Prova: *supomos que ab é ímpar. Desejamos mostrar que a é ímpar ou que b é par. Então tentaremos obter uma contradição assumindo o oposto, ou seja, que a é par e b é ímpar. Se for assim, então* $a = 2i$ *para algum inteiro i e portanto* $ab = (2i)b = 2(ib)$*, ou seja, ab é par. Mas começamos supondo que ab é ímpar e agora concluímos que ab é par. Obtemos uma contradição, e portanto a é ímpar ou b é par.* ■

Indução

A maior parte das afirmações que fazemos sobre o tempo de execução de um algoritmo diz respeito a um parâmetro inteiro n (em geral representando uma noção intuitiva do "tamanho" do problema). Além disso, a maior parte dessas afirmações equivale a dizer que determinada afirmação $q(n)$ é verdadeira "para todo $n \geq 1$". Como isso equivale a fazer uma afirmação sobre um conjunto infinito de números, não podemos justificá-la de forma exaustiva. Entretanto, podemos freqüentemente justificar afirmações como as feitas acima se fizermos uso da técnica da ***indução***. Essa técnica resume-se em mostrar que para qualquer $n \geq 1$ existe uma seqüência finita de implicações que inicia com um fato verdadeiro e leva à confirmação de que $q(n)$ é verdadeiro. Especificamente, começamos uma justificativa por indução mostrando que $q(n)$ é verdadeiro para $n = 1$ (e possivelmente outros valores $n = 2,3, ..., k$ para alguma constante k). Depois justificamos que o "passo" indutivo é verdadeiro para $n > k$, ou seja, mostramos que "se $q(i)$ é verdadeiro para $i < n$ então $q(n)$ é verdadeiro". A combinação destas duas partes completa a justificativa por indução.

Exemplo 1.19: *considere a seqüência de Fibonacci:* $F(1) = 1$, $F(2) = 2$, *e* $F(n) = F(n-1) + F(n-2)$ *para* $n > 2$. *Afirmamos que* $F(n) < 2^n$.

Prova: *faremos a justificativa por indução.*
Caso base: $(n \leq 2)$. $F(1) = 1 < 2 = 2^1$ *e* $F(2) = 2 < 4 = 2^2$.
Passo da indução: $(n > 2)$. *Suponha que a afirmação é verdadeira para* $n' < n$. *Considere* $F(n)$. *Já que* $n > 2$, *então* $F(n) = F(n-1) + F(n-2)$. *Além disso, já que* $n - 1 < n$ *e* $n - 2 < n$, *podemos aplicar a suposição da indução (às vezes chamada de hipótese de indução) para implicar que* $F(n) < 2^{n-1} + 2^{n-2}$. *Mas*

$$2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n$$

Isso completa a prova. ■

Vamos fazer outro argumento indutivo, desta vez para um fato já visto antes.

Teorema 1.20: (*que equivale ao Teorema 1.13*)

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Prova: *faremos a justificativa por indução.*
Caso base: $n = 1$. *É trivial, pois* $1 = n(n+1)/2$, *se* $n = 1$.
Passo da indução: $n \geq 2$. *Suponha que a afirmação é verdadeira para* $n' < n$, *e considere* n. Então,

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i.$$

Pela hipótese de indução, temos

$$\sum_{i=1}^{n} i = n + \frac{(n-1)n}{2},$$

que podemos simplificar como

$$n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}.$$

Isso completa a justificativa. ■

Podemos às vezes nos sentir sobrecarregados pela tarefa de justificar algo verdadeiro para todo $n \geq 1$. Deveríamos lembrar, no entanto, que a técnica de indução é bastante concreta: ela mostra que para qualquer n em particular existe uma seqüência de implicações que se inicia com um fato verdadeiro e leva a uma verdade sobre n. Resumindo, o argumento indutivo é uma fórmula para construir uma seqüência de justificativas diretas.

Invariantes de laços

A técnica final de justificativa que discutiremos é o uso de ***invariantes de laços***, que é usada para analisar e provar a correção de laços.

Para provar que uma afirmação S sobre um laço é correta, defina S como uma seqüência de afirmações menores $S_0, S_1, \ldots, S_k$, onde:

1. A afirmação ***inicial*** S_0 é verdadeira antes que o laço se inicie.
2. Se S_{i-1} é verdadeira antes que a iteração i se inicie, então pode-se mostrar que S_i será verdadeira depois que a iteração i terminar.
3. A afirmação final S_k implica que a afirmação S que desejávamos provar é verdadeira.

De fato, já vimos a técnica da justificativa por invariantes de laços em operação na Seção 1.1.1 (discutindo a correção do algoritmo arrayMax), e agora veremos mais um exemplo. Em particular, vamos usar o método para mostrar a correção do algoritmo arrayFind, mostrado no Algoritmo 1.12, que resolve o problema de encontrar um elemento x em um arranjo A.

Para mostrar que o algoritmo arrayFind é correto, argumentaremos através da invariante do laço, ou seja, definiremos uma série de afirmações S_i que demonstrarão a correção do algoritmo. Especificamente, afirmamos que o seguinte fato é verdadeiro no início da iteração i:

S_i: x não é igual a nenhum dos primeiros i elementos de A.

Esta afirmação é verdadeira no início da primeira iteração do laço, já que não há elementos entre os primeiros 0 elementos de A. Na iteração i, comparamos o elemento x com o elemento $A[i]$ e retornamos o índice i se eles forem iguais, o que é clara-

mente correto e completa o algoritmo neste caso. Se os elementos x e $A[i]$ não são iguais, então teremos achado mais um elemento diferente de x e incrementamos o índice i. Assim, a afirmação S_i será verdadeira para este novo valor de i e, portanto, será verdadeira no início da próxima iteração. Se o laço **enquanto** termina sem nunca retornar um índice em A, então devemos ter $i = n$. Ou seja, S_n é verdadeiro – não há elementos em A que sejam iguais a x. Portanto, o algoritmo está correto em retornar o valor –1, que é inválido como índice, como solicitado.

Algoritmo arrayFind(x, A):
  Entrada: um elemento x e um arranjo A com n elementos.
  Saída: o índice i tal que $x = A[i]$, ou –1 se x não está em A.
$i \leftarrow 0$
enquanto $i < n$ **faça**
  se $x = A[i]$ **então**
    retorne i
  senão
    $i \leftarrow i + 1$
retorne –1

Algoritmo 1.12 Algoritmo arrayFind.

1.3.4 Probabilidade básica

Quando analisamos algoritmos que fazem uso de randomização ou se desejamos analisar o desempenho de um algoritmo no caso médio, precisamos de alguns conhecimentos básicos de teoria das probabilidades. O fato mais básico é que qualquer afirmação sobre probabilidades é definida sobre um ***espaço amostral*** S, definido como o conjunto de todos os possíveis resultados de um experimento. Deixaremos os termos "resultado" e "experimento" indefinidos em um sentido formal.

Exemplo 1.21: *considere um experimento que consiste em jogar uma moeda cinco vezes. Este espaço amostral tem* 2^5 *resultados diferentes, um para cada ordem diferente dos resultados das moedas jogadas individualmente.*

Espaços amostrais podem ser infinitos, como ilustrado no exemplo a seguir.

Exemplo 1.22: *considere um experimento que consiste em jogar uma moeda até que ela dê cara. Este espaço amostral é infinito, com cada saída sendo uma seqüência de* i *coroas seguidas por uma cara, para* $i \in \{0,1,2,3, ...\}$.

Um ***espaço de probabilidade*** é um espaço amostral S com uma função de probabilidade Pr que mapeia subconjuntos de S a números reais no intervalo [0,1]. Ele captura matematicamente a noção da probabilidade de certos "eventos" ocorrerem. Formalmente, cada subconjunto A de S é chamado um ***evento***, e a função de probabilidade Pr possui as seguintes propriedades básicas com respeito a eventos definidos em S:

1. $Pr(\emptyset) = 0$.
2. $Pr(S) = 1$.

3. $0 \leq Pr(A) \leq 1$, para todo $A \subseteq S$.
4. Se $A, B \subseteq S$ e $A \cap B = \emptyset$, então $Pr(A \cup \mathrm{B}) = Pr(A) + Pr(B)$.

Independência

Dois eventos A e B são ***independentes*** se

$$Pr(A \cap B) = Pr(A) \cdot \mathrm{Pr}(B).$$

Uma coleção de eventos $\{A_1, A_2, ..., A_n\}$ é ***mutuamente independente*** se

$$Pr(A_{i_1} \cap A_{i_2} \cdots \cap A_{i_k}) = Pr(A_{i_1})\, Pr(A_{i_2}) \cdots Pr(A_{i_k}).$$

para qualquer subconjunto $\{A_{i_1}, A_{i_2}, ..., A_{i_k}\}$.

Exemplo 1.23: *seja A o evento de um dado lançado dar o número* 6, *B o evento de um segundo dado dar o número* 3 *e C o evento da soma destes dois dados dar o valor* 10. *Então A e B são eventos independentes, mas C não é independente de A ou de B.*

Probabilidade condicional

A ***probabilidade condicional*** de que um evento A ocorra, dado um evento B, é denotada como $Pr(A|B)$ e definida como

$$Pr(A \mid B) = \frac{Pr(A \cap B)}{Pr(B)}$$

supondo que $Pr(B) > 0$.

Exemplo 1.24: *seja A o evento do lançamento de dois dados somar* 10 *pontos, e seja B o evento do primeiro dado dar valor* 6. *Note que* $Pr(B) = 1/6$, *e que* $Pr(A \cap B) = 1/36$, *pois só há uma maneira de dois dados somarem* 10 *pontos se o primeiro dado deu* 6 *(naturalmente, o segundo dado deve dar* 4*). Assim,* $Pr(A|B) = (1/36)/(1/6) = 1/6$.

Variáveis aleatórias e expectância

Uma maneira elegante de lidar com eventos é em termos de ***variáveis aleatórias***. Intuitivamente, variáveis aleatórias são variáveis cujos valores dependem do resultado de algum experimento. Formalmente, uma ***variável aleatória*** é uma função X que mapeia resultados de um espaço amostral S a números reais. Uma ***variável aleatória indicadora*** é uma variável aleatória que mapeia resultados para os valores 0 e 1. Freqüentemente na análise de estruturas de dados e algoritmos usamos uma variável aleatória X para caracterizar o tempo de execução de um algoritmo randomizado. Neste caso, o espaço de amostragem em S é definido por todos os resultados aleatórios possíveis usados no algoritmo. Estamos mais interessados no valor típico, médio ou "esperado" de uma variável aleatória. O ***valor esperado*** de uma variável aleatória X é definido como

$$E(X) = \sum_x xPr(X = x),$$

onde a soma é definida sobre todos os valores em X.

Teorema 1.25: (linearidade do valor esperado) *sejam X e Y duas variáveis aleatórias arbitrárias. Então* $E(X + Y) = E(X) + E(Y)$.

Prova:

$$\begin{aligned}
E(X+Y) &= \sum_x \sum_y (x+y) Pr(X = x \cap Y = y), \\
&= \sum_x \sum_y xPr(X = x \cap Y = y) + \sum_x \sum_y yPr(X = x \cap Y = y) \\
&= \sum_x \sum_y xPr(X = x \cap Y = y) + \sum_x \sum_y yPr(Y = y \cap X = x) \\
&= \sum_x xPr(X = x) + \sum_y yPr(Y = y) \\
&= E(X) + E(Y).
\end{aligned}$$

Note que essa prova não depende de quaisquer hipóteses de independência entre os eventos quando X e Y assumem seus valores. ■

Exemplo 1.26: *seja X uma variável aleatória que associa o resultado do lançamento de dois dados à soma dos valores resultantes. Então* $E(X) = 7$.

Prova: *para justificar a afirmação, sejam* X_1 *e* X_2 *variáveis aleatórias correspondendo ao número resultante em cada dado. Assim,* $X_1 = X_2$ *(isto é, são duas instâncias da mesma função) e* $E(X) = E(X_1 + X_2) = E(X_1) + E(X_2)$. *Cada resultado do lançamento dos dados ocorre com probabilidade* 1/6. *Portanto*

$$E(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2},$$

para $i = 1,2$. *Por isso,* $E(X) = 7$. ■

Duas variáveis aleatórias X e Y são independentes se

$$Pr(X = x/Y = y) = Pr(X = x),$$

para todos os números x e y.

Teorema 1.27: *se duas variáveis aleatórias X e Y são independentes, então*

$$E(XY) = E(X)E(Y)$$

Exemplo 1.28: *seja X uma variável aleatória relacionando o resultado do lançamento de dois dados ao produto dos valores resultantes. Então* $E(X) = 49/4$.

Prova: *sejam* X_1 *e* X_2 *variáveis aleatórias correspondendo ao número resultante em cada dado. As variáveis* X_1 *e* X_2 *são claramente independentes, portanto*

$$E(X) = E(X_1X_2) = E(X_1)E(X_2) = (7/2)^2 = 49/4.$$

■

Limites de Chernoff

Na análise de algoritmos randomizados, é freqüentemente necessário obter um limite para a soma de um conjunto de variáveis aleatórias. Um conjunto de desigualdades que torna esse problema manejável são os Limites de Chernoff. Sejam $X_1, X_2, ..., X_n$ um conjunto de variáveis aleatórias mutuamente independentes, tais que cada X_i é 1 com probabilidade $p_i > 0$ e 0 em caso contrário. Seja $X = \sum_{i=1}^{n} X_i$ a soma dessas variáveis aleatórias, e seja μ a média de X, ou seja, $\mu = E(X) = \sum_{i=1}^{n} p_i$. Damos o resultado a seguir, sem prová-lo.

Teorema 1.29: *seja X como acima. Então, para* $\delta > 0$,

$$Pr(X > (1+\delta)\mu) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu},$$

e, para $0 < \delta \leq 1$,

$$Pr(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}.$$

1.4 Exemplos de análise assintótica de algoritmos

Tendo apresentado a metodologia geral para descrição e análise de algoritmos, consideramos agora alguns exemplos na análise de algoritmos. Especificamente, mostramos como usar a notação O para analisar dois algoritmos que resolvem o mesmo problema, mas têm tempos de execução diferentes.

O problema em questão é fazer o cálculo das ***médias prefixadas*** de uma seqüência de números. Ou seja, se tivermos um arranjo X armazenando n números, desejamos compor um arranjo A tal que $A[i]$ é a média dos elementos $X[0], X[1], ..., X[i]$, para $i = 0,..., n-1$. Ou seja,

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

As médias prefixadas têm várias aplicações em economia e estatística. Por exemplo, dados os retornos anuais de um fundo de investimento, podemos avaliar o retorno médio anual do fundo no último ano, nos últimos três, cinco ou dez anos. A média prefixada também é útil como função "suavizadora" para um parâmetro que varia rapidamente, como ilustrado na Figura 1.13.

Um algoritmo de tempo quadrático

Nosso primeiro algoritmo para o problema das médias prefixadas, chamado de prefixAverages1, é mostrado no Algoritmo 1.14. Ele calcula cada elemento de A separadamente, de acordo com a definição.

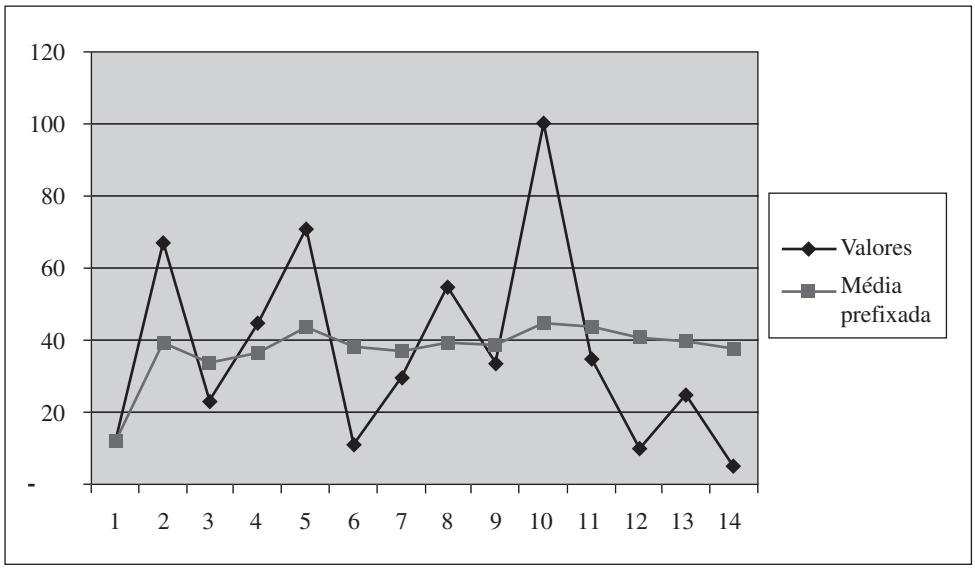


Figura 1.13 Uma ilustração da média prefixada e seu uso como função suavizadora de uma seqüência de valores que oscilam rapidamente.

Algoritmo prefixAverages1(X):
  Entrada: um arranjo X com n elementos.
  Saída: um arranjo A com n elementos tal que $A[i]$ é
    a média de $X[0], X[1],..., X[i]$.
Seja A um arranjo de n números.
para $i \leftarrow 0$ até $n - 1$ **faça**
    $a \leftarrow 0$
    para $j \leftarrow 0$ até i **faça**
      $a \leftarrow a + X[j]$
    $A[i] \leftarrow a/(i + 1)$
retorne array A

Algoritmo 1.14 Algoritmo prefixAverages1.

Vamos analisar o algoritmo prefixAverages1.

- Inicializar o arranjo A no início e retorná-lo ao final são ações que custam um número constante de operações primitivas por elemento de A, e custam tempo $O(n)$.
- Existem dois laços **para** aninhados, controlados por contadores i e j. O corpo do laço externo (controlado por i) é executado n vezes, para valores $i = 0,..., n - 1$. Assim, os comandos $a = 0$ e $A[i] = a/(i + 1)$ são executados n vezes cada. Isso implica que esses dois comandos, mais o incremento e teste do contador i, contribuem com um número de operações primitivas proporcional a n, ou seja, $O(n)$.

- O corpo do laço interno (controlado por j) é executado $i + 1$ vezes, dependendo do contador i do laço externo. Assim, o comando $a = a + X[j]$ é executado $1 + 2 + 3 + \cdots + n$ vezes. Pelo Teorema 1.13, sabemos que $1 + 2 + 3 + \cdots + n = n(n + 1)/2$, o que implica que o comando do laço interno contribui com tempo $O(n^2)$. Um argumento similar pode ser feito para as operações primitivas associadas ao incremento e teste de j, que também custam tempo $O(n^2)$.

O tempo de execução de prefixAverages1 é dado pela soma dos três termos. O primeiro e o segundo termos são $O(n)$ e o terceiro é $O(n^2)$. Aplicando o Teorema 1.7, temos que o tempo de execução de prefixAverages1 é $O(n^2)$.

1.4.2 Um algoritmo de tempo linear

Para calcular ***médias prefixadas*** mais eficientemente, podemos observar que duas médias consecutivas $A[i]$ e $A[i + 1]$ são similares:

$$A[i - 1] = (X[0] + X[1] + \cdots + X[i - 1])/i$$
$$A[i] = (X[0] + X[1] + \cdots + X[i - 1] + X[i])/(i + 1).$$

Se denotarmos com S_i a ***soma prefixada*** $X[0] + X[1] + \cdots + X[i]$, podemos calcular as ***médias prefixadas*** como sendo $A[i] = S_i / (i + 1)$. É fácil manter o controle da soma prefixada corrente enquanto varremos o arranjo X em um laço. Estamos prontos para apresentar o algoritmo prefixAverages2 no Algoritmo 1.15.

Algoritmo prefixAverages2(X):
  Entrada: um arranjo X com n elementos.
  Saída: um arranjo A com n elementos tal que $A[i]$ é a média de $X[0], \ldots X[i]$.
Seja A um arranjo de n números.
$s \leftarrow 0$
para $i \leftarrow 0$ **até** $n - 1$ **faça**
  $s \leftarrow s + X[i]$
  $A[i] \leftarrow s/(i + 1)$
retorne array A

Algoritmo 1.15 Algoritmo para prefixAverages2.

A análise do tempo de execução do algoritmo prefixAverages2 vem a seguir:

- Inicializar o arranjo A no início e retorná-lo ao final são ações que custam um número constante de operações primitivas por elemento de A, e custam tempo $O(n)$.
- Inicializar a variável s no início custa tempo $O(1)$.
- Existe um laço **para** controlado por um contador i. O corpo do laço é executado n vezes, para valores $i = 0,..., n - 1$. Assim, os comandos $s = s + X[i]$ e $A[i] = s/(i + 1)$ são executados n vezes cada. Isso implica que esses dois comandos, mais o incremento e teste do contador i, contribuem com um número de operações primitivas proporcional a n, ou seja, $O(n)$.

O tempo de execução de prefixAverages2 é dado pela soma dos três termos. O primeiro e o terceiro termos são $O(n)$, e o segundo é $O(1)$. Aplicando o Teorema 1.7, temos

que o tempo de execução de prefixAverages2 é $O(n)$, muito melhor do que o tempo quadrático de prefixAverages1.

1.5 Amortização

Uma ferramenta de análise importante para a compreensão dos tempos de execução de algoritmos que têm trechos onde o desempenho pode variar muito é a ***amortização***. O termo "amortização" origina-se da contabilidade, que provê uma metáfora intuitiva para a análise de algoritmos, como veremos nesta seção.

Uma estrutura de dados típica geralmente suporta uma variedade de métodos diferentes para acesso e atualização dos elementos armazenados. Da mesma forma, alguns algoritmos operam iterativamente, com cada iteração realizando uma quantidade de trabalho diferente. Em alguns casos, podemos analisar efetivamente o desempenho dessas estruturas de dados e algoritmos com base no tempo de execução de pior caso de cada operação individual. A amortização apresenta um ponto de vista diferente. Em vez de concentrar-se em cada operação individualmente, ela considera as interações entre todas as operações estudando os tempos de execução de uma série destas operações.

A tabela apagável

Como exemplo, vamos introduzir um tipo abstrato de dados (TAD), a ***tabela apagável***. Esse TAD armazena uma tabela de elementos que podem ser acessados por seu índice na tabela. Além disso, a tabela apagável também suporta os dois métodos seguintes:

add(e): adiciona o elemento e na próxima posição disponível na tabela.

clear( ): esvazia a tabela removendo todos os elementos.

Seja S uma tabela apagável com n elementos implementada como um arranjo, com um limite superior N para seu tamanho. A operação clear toma tempo $\Theta(n)$, pois devemos apagar todos os elementos na tabela para esvaziá-la.

Agora vamos considerar uma série de n operações em uma tabela apagável S, inicialmente vazia. Se usarmos o ponto de vista do pior caso, podemos dizer que o tempo de execução desta série de operações é $O(n^2)$, pois o pior caso de uma única operação clear é $O(n)$ e existem $O(n)$ operações clear na seqüência. Enquanto essa análise é correta, ela também é exagerada, pois uma análise que leva em conta as interações entre as operações mostra que o tempo de execução da seqüência completa é $O(n)$.

Teorema 1.30: *uma série de n operações em uma tabela apagável inicialmente vazia implementada com um arranjo custa tempo $O(n)$.*

Prova: seja $M_0,\ldots,M_{n-1}$ a seqüência de operações realizadas sobre S, e sejam $M_{i_0},\ldots,M_{i_{k-1}}$ as k operações de clear na seqüência. Temos

$$0 \le i_0 < \ldots < i_{k-1} \le n-1.$$

Vamos também definir $i_{-1} = -1$. O tempo de execução da operação M_{i_j} (uma operação clear) é $O(i_j - i_{j-1})$, pois no máximo $i_j - i_{j-1} - 1$ elementos podem ter sido adicionados

na tabela usando-se a operação add desde a última operação clear (em $M_{i_{j-1}}$) ou desde o começo da seqüência. Assim, o tempo de execução de todas as operações clear é

$$O\left(\sum_{j=0}^{k-1}(i_j - i_{j-1})\right).$$

Uma soma como essa é conhecida como uma ***soma telescópica***, pois todos os termos exceto o primeiro e o último se cancelam. Ou seja, esta soma é $O(i_{k-1} - i_{-1})$, que é $O(n)$. As operações restantes da seqüência custam tempo $O(1)$ cada uma. Assim, concluímos que uma série de n operações realizadas em uma tabela apagável inicialmente vazia custa tempo $O(n)$. ■

O Teorema 1.30 indica que o tempo de execução médio de qualquer operação em uma tabela apagável é $O(1)$, onde a média é feita sobre uma série arbitrária de operações iniciando-se com uma tabela vazia.

Amortizando o tempo de execução de um algoritmo

O exemplo anterior fornece a motivação para a técnica de amortização, que provê uma forma de análise de caso médio baseada no pior caso. Formalmente, definimos o ***tempo de execução amortizado*** de uma operação em uma seqüência de operações como o tempo de execução de pior caso da seqüência de operações dividido pelo número de operações. Quando a seqüência de operações não é especificada, é geralmente presumida como uma seqüência de operações dentre as possíveis para uma dada estrutura de dados e iniciando com uma estrutura vazia. Assim, pelo Teorema 1.30, podemos dizer que o tempo de execução amortizado de cada operação no TAD tabela apagável é $O(1)$ quando a implementamos com um arranjo. Note que o tempo de execução real de uma operação pode ser muito mais alto do que seu tempo de execução amortizado (por exemplo, uma operação clear em particular pode ter tempo $O(n)$).

A vantagem de se usar amortização é que ela fornece uma análise robusta de caso médio sem uso de probabilidade. Ela apenas exige que tenhamos uma maneira de caracterizar o tempo de execução de pior caso de uma seqüência de operações. Podemos mesmo estender a noção de tempo de execução amortizado e associar a cada operação individual, em uma seqüência de operações, seu próprio tempo de execução amortizado, desde que o tempo total necessário para processar toda a seqüência de operações não seja maior do que a soma dos tempos amortizados das operações individuais.

Existem várias maneiras de fazer a análise amortizada. A maneira mais óbvia é usar um argumento direto para obter limites para o tempo total necessário para realizar uma seqüência de operações, e foi o que fizemos no Teorema 1.30. Enquanto argumentos diretos podem freqüentemente ser obtidos por uma série de operações simples, realizar uma análise amortizada de uma seqüência não-trivial de operações é geralmente mais fácil se usarmos técnicas especiais de análise amortizada.

1.5.1 Técnicas de amortização

Existem duas técnicas fundamentais para realizar uma análise amortizada: uma baseada em uma metáfora financeira – o método da contabilidade – e o outra baseada em um modelo de energia – o método da função potencial.

O método da contabilidade

O ***método da contabilidade***, para realizar a análise amortizada, usa um sistema de créditos e débitos para manter o tempo de execução das diferentes operações da seqüência. A base do método da contabilidade é simples. Vemos o computador como uma máquina que funciona com moedas e requer o pagamento de um ***ciberdólar*** para um tempo constante de computação. Também vemos uma operação como uma seqüência de ***operações primitivas*** de tempo constante, cada uma delas custando um ***ciberdólar*** para ser executada. Quando uma operação é executada, devemos ter ciberdólares suficientes para pagar por seu tempo de execução. A abordagem mais óbvia é cobrar para uma operação o mesmo número de operações primitivas que ela precisa para ser realizada. Entretanto, o aspecto interessante do método da contabilidade é que não precisamos agir assim na cobrança pelas operações. Ou seja, podemos cobrar mais caro por algumas operações que executam poucas operações primitivas e usar o lucro para financiar outras operações que executam muitas operações primitivas. Esse mecanismo pode permitir-nos cobrar a mesma quantia a de ciberdólares para cada operação na seqüência sem ficar sem ciberdólares para pagar o tempo de computador. Assim, se pudermos montar um ***sistema de amortização***, podemos dizer que cada operação na série tem um tempo de execução amortizado que é $O(a)$. Quando projetamos um sistema de amortização, é freqüentemente conveniente pensar nos ciberdólares não gastos como se estivessem armazenados em certos lugares da estrutura de dados, por exemplo, nos elementos de uma tabela.

Um sistema de amortização alternativo cobra quantias diferentes pelas várias operações. Neste caso, o tempo de execução amortizado de uma operação é proporcional à cobrança total dividida pelo número de operações.

Voltamos ao exemplo da tabela apagável e apresentamos um sistema de amortização que fornece uma prova alternativa para o Teorema 1.30. Vamos supor que um ciberdólar é suficiente para pagar pela execução da operação de acesso a um índice ou uma operação add e também pelo tempo usado na operação clear para destruir um único elemento, e por isso cobraremos dois ciberdólares em cada operação. Ou seja, estamos cobrando muito barato pela operação clear e um ciberdólar a mais pelas outras operações. O ciberdólar que lucramos em uma operação add fica armazenado no elemento inserido pela operação (veja a Figura 1.16.). Quando uma operação clear é executada, o ciberdólar armazenado em cada elemento da tabela é usado para pagar o tempo gasto destruindo-o. Assim, temos um sistema válido de amortização onde cada operação custa dois ciberdólares, e todo o tempo de computação é pago. Esse sistema simples de amortização implica o resultado do Teorema 1.30.

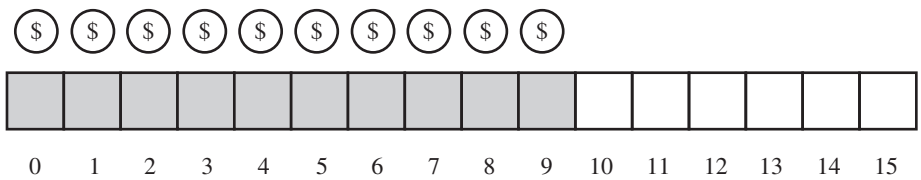


Figura 1.16 Ciberdólares armazenados nos elementos de uma tabela apagável S na análise amortizada de uma série de operações em S.

Note que o pior caso para o tempo de execução ocorre para uma série de operações add seguidas por uma operação clear. Em outros casos, ao final da série de operações podemos terminar com alguns ciberdólares armazenados, que foram os obtidos por acessos a elementos e os armazenados nos elementos ainda na tabela. De fato, o tempo de execução para executar uma série de n operações pode ser pago com um total de ciberdólares entre n e $2n$. Nosso sistema de amortização contabiliza para o pior caso, pois sempre cobra dois ciberdólares por operação.

Funções de potencial

Outra técnica útil para realizar a análise amortizada é baseada em um modelo de energia. Nesta abordagem, associamos a nossa estrutura um valor Φ, que representa o estado atual de energia de nosso sistema. Cada operação que realizamos contribui para Φ com uma quantidade adicional chamada de tempo amortizado, mas também diminui Φ de forma proporcional ao tempo total realmente gasto. Formalmente, deixamos $\Phi_0 \geq 0$ denotar o valor inicial de Φ antes de realizarmos qualquer operação, e usamos Φ_i para denotar o valor da função de potencial Φ depois de realizarmos a i-ésima operação. A idéia principal na função de potencial é usar a alteração em potencial na i-ésima operação, $\Phi_i - \Phi_{i-1}$ para caracterizar o tempo amortizado necessário para a operação.

Considerando a ação da i-ésima operação, fazemos t_i denotar seu tempo de execução. Definimos o tempo de execução amortizado da i-ésima operação como

$$t_i' = t_i + \Phi_i - \Phi_{i-1}.$$

Ou seja, o custo amortizado da i-ésima operação é o tempo de execução mais a alteração em potencial que a operação causa (que pode ser positivo ou negativo). Colocando de outra forma,

$$t_i = t_i' + \Phi_{i-1} - \Phi_i.$$

ou seja, o tempo realmente gasto é o custo amortizado mais a alteração em potencial.

Denote como T' o tempo total amortizado para realizar n operações em nossa estrutura. Ou seja,

$$T' = \sum_{i=1}^{n} t_i'.$$

Então o tempo total real T usado pelas n operações pode ser limitado por

$$\begin{aligned}
T &= \sum_{i=1}^{n} t_i \\
&= \sum_{i=1}^{n} (t_i' + \Phi_{i-1} - \Phi_i) \\
&= \sum_{i=1}^{n} t'_i{}_i + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i) \\
&= T' + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i) \\
&= T' + \Phi_0 - \Phi_n,
\end{aligned}$$

já que o segundo termo forma uma soma telescópica. Em outras palavras, o tempo total gasto é igual ao tempo total amortizado mais a alteração de potencial na seqüência de operações. Assim, enquanto $\Phi_n \geq \Phi_0$, então $T \leq T'$, e o tempo total real não é maior do que o tempo amortizado.

Para tornar esse conceito mais concreto, vamos repetir a análise da tabela apagável usando esse tipo de argumento. Neste caso, escolhemos o potencial Φ do sistema como sendo o número de elementos na tabela. Afirmamos que o tempo amortizado para qualquer operação é 2, ou seja, $t'_i = 2$ para $i = 1,..., n$. Para justificar, vamos considerar os dois métodos possíveis para a i-ésima operação.

- add(e): inserir o elemento e na tabela aumenta Φ em 1, e o tempo real usado é uma unidade de tempo. Assim, neste caso

$$1 = t_i = t'_i + \Phi_{i-1} - \Phi_i = 2 - 1,$$

  o que é claramente verdadeiro.

- clear( ): remover os m elementos da tabela requer não mais do que $m + 2$ unidades de tempo — m unidades para a remoção e 2 unidades no máximo para a chamada do método e seu custo interno. Essa operação também muda o potencial Φ do sistema de m para 0. Assim, neste caso

$$m + 2 = t_i = t'_i + \Phi_{i-1} - \Phi_i = 2 + m,$$

  o que também é claramente verdadeiro.

Portanto, o tempo amortizado para realizar qualquer operação em uma tabela apagável é $O(1)$. Além disso, já que $\Phi_i \geq \Phi_0$ para qualquer $i \geq 1$, o tempo real T para realizar n operações em uma tabela inicialmente vazia é $O(n)$.

1.5.2 Análise de uma implementação de arranjo expansível

Uma fraqueza da implementação de arranjos para a tabela apagável dada acima é que ela requer a especificação prévia de uma capacidade fixa N para o número total de elementos que podem ser armazenados na tabela. Se o número real de elementos n da tabela é muito menor do que N, então a implementação desperdiça espaço. Pior, se n for maior do que N, a implementação terá problemas.

Vamos prover um meio de aumentar o arranjo A que armazena os elementos da tabela S. É claro que, em qualquer linguagem de programação convencional como C, C++ ou Java, não podemos realmente aumentar o arranjo A; sua capacidade é fixa em um número N. Em vez disso, quando um ***overflow*** acontece, ou seja, quando $n = N$ e o método add é chamado, realizamos os passos a seguir:

1. alocamos um novo arranjo B de capacidade $2N$;
2. copiamos $A[i]$ para $B[i]$, para $i = 0,..., N - 1$;
3. fazemos $A = B$, ou seja, usamos B como o arranjo implementando S.

Essa estratégia de substituição do arranjo é conhecida como ***arranjo expansível*** (veja a Figura 1.17). Intuitivamente, essa estratégia é como a de um caranguejo, que muda de casca quando a antiga está pequena demais.

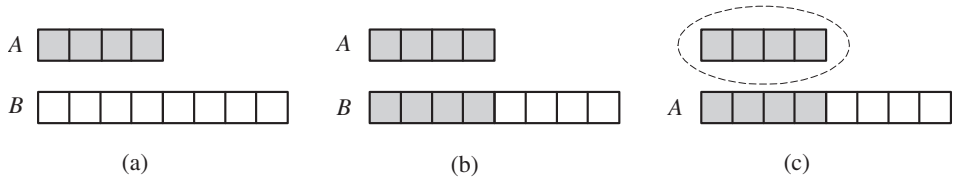


Figura 1.17 Ilustração dos três passos para aumentar um arranjo expansível: (a) criar o novo arranjo *B*; (b) copiar elementos de *A* para *B*; (c) usar *B* como o novo arranjo. Não são mostradas a coleta de lixo e a reutilização do antigo arranjo *A*.

Em termos de eficiência, essa estratégia de substituição de arranjos pode parecer lenta, pois substituir um arranjo de tamanho n exige tempo $\Theta(n)$. Note que, depois de realizarmos a substituição, nosso novo arranjo permite que adicionemos n novos elementos na tabela antes que o arranjo tenha de ser substituído outra vez. Esse fato permite-nos mostrar que o tempo de execução de uma série de operações realizadas em uma tabela expansível inicialmente vazia é bastante eficiente. Para facilitar, vamos chamar a inserção de um elemento na última posição de um vetor como uma operação de "add". Usando ***amortização***, podemos mostrar que realizar uma seqüência de operações add em uma tabela implementada com um arranjo expansível é bastante eficiente.

Teorema 1.31: *seja S uma tabela implementada com um arranjo expansível A descrito como acima. O tempo total para realizar uma seqüência de n operações de* add *em S, iniciando com S vazia e A tendo tamanho* $N = 1$, *é* $O(n)$.

Prova: justificamos esse teorema usando o método de contabilidade para ***amortização***. Para esta análise, vemos o computador como uma máquina que funciona com moedas e requer o pagamento de um ***ciberdólar*** para um tempo constante de computação. Quando uma operação é executada, devemos ter ciberdólares suficientes em nossa "conta corrente" para pagar por seu tempo de execução. Assim, o total de ciberdólares gasto em uma computação será proporcional ao tempo total gasto na computação. A beleza desta análise reside no fato de que podemos cobrar mais por algumas operações para pagar por outras.

Vamos supor que um ciberdólar é suficiente para pagar pela execução de cada operação de add em S, excluindo o tempo para expandir o arranjo. Também vamos supor que aumentar o arranjo do tamanho k para o tamanho $2k$ requer k ciberdólares pelo tempo gasto copiando os elementos. Cobraremos três ciberdólares por uma operação de add. Assim, cobraremos dois ciberdólares a mais por uma operação de add que não causa *overflow*. Vamos pensar nestes dois ciberdólares como estando armazenados junto ao elemento inserido. Um *overflow* ocorre quando a tabela S tem 2^i elementos para algum $i \geq 0$, e o tamanho do arranjo usado por S é 2^i. Assim, dobrar o tamanho do arranjo requer 2^i ciberdólares. Felizmente, esses ciberdólares podem ser encontrados nos elementos armazenados nas posições 2^{i-1} a $2^i - 1$ (veja a Figura 1.18). Note que o *overflow* anterior ocorreu quando o número de elementos foi maior do que 2^{i-1} pela primeira vez, e os ciberdólares armazenados nas posições entre 2^{i-1} e $2^i - 1$ ainda não tinham sido gastos. Portanto, temos um sistema válido de amortização no qual cobramos três ciberdólares por operação e todo o tempo de computação é pago. Ou seja, podemos pagar pela execução de n operações add usando $3n$ ciberdólares. ■

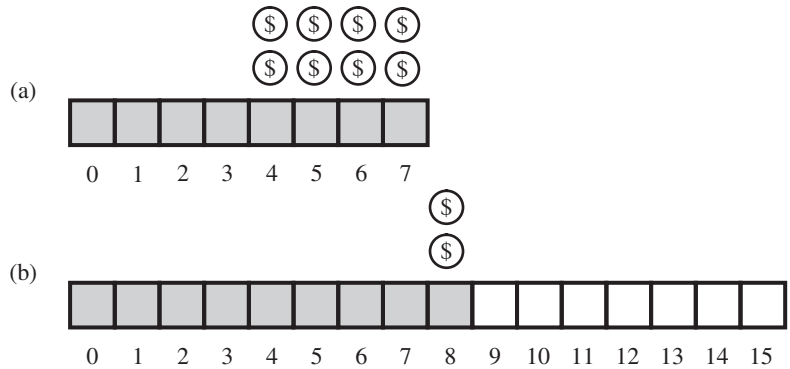


Figura 1.18 Uma operação add cara: (a) um arranjo de oito posições cheio, com dois ciberdólares em cada posição de 4 a 7; (b) uma operação add dobra a capacidade. Copiar elementos gasta os ciberdólares da tabela, inserir o novo elemento gasta um ciberdólar da operação add, e os dois ciberdólares restantes são armazenados na célula 8.

Uma tabela pode dobrar de tamanho a cada expansão, como descrevemos, ou podemos especificar uma quantidade específica, chamando-a de capacityIncrement, como parâmetro que determina o quanto o vetor cresce a cada expansão. Ou seja, se esse parâmetro vale k, o arranjo aumenta em k unidades quando é expandido. No entanto, devemos usar um parâmetro desses com cautela. Para a maior parte das aplicações, dobrar o tamanho é a escolha certa, como mostra o teorema abaixo.

Teorema 1.32: *se criarmos uma tabela inicialmente vazia com um valor fixo e positivo para* capacityIncrement, *realizar uma série de n operações* add *na tabela custa tempo* $\Omega(n^2)$.

Prova: seja $c > 0$ o valor de capacityIncrement e seja $c_0 > 0$ o tamanho inicial do arranjo. Um *overflow* será causado por uma operação de add quando o número corrente

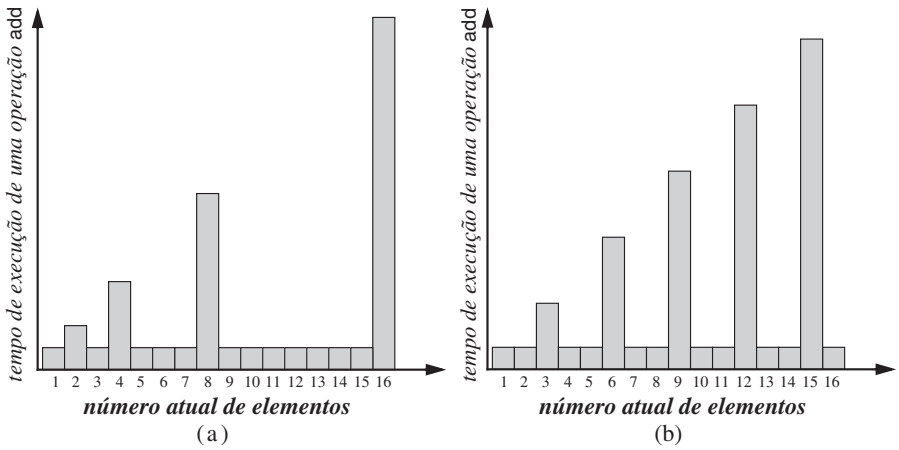


Figura 1.19 Tempos de execução de uma seqüência de operações de add em uma tabela expansível. Em (a) o tamanho é dobrado a casa expansão, e em (b) ele é incrementado com capacityIncrement = 3.

de elementos na tabela for de $c_0 + ic$, para $i = 0,..., m - 1$, onde $m = \lfloor (n - c_0)/c \rfloor$. Assim, pelo Teorema 1.13, o tempo total para tratar o *overflow* é proporcional a

$$\sum_{i=0}^{m-1}(c_0 + ci) = c_0 m + c\sum_{i=0}^{m-1} i = c_0 m + c\frac{m(m-1)}{2},$$

que é $\Omega(n^2)$. Portanto, realizar as n operações de add custa tempo $\Omega(n^2)$. ■

A Figura 1.19 compara os tempos de execução de uma série de operações add, em uma tabela inicialmente vazia, para dois valores de capacityIncrement.

Discutiremos aplicações de amortização quando discutirmos árvores *splay* (Seção 3.4) e uma estrutura de árvore para realizarmos a união e a pesquisa em partições de conjuntos (Seção 4.2.2).

1.6 Experimentação

Usar análise assintótica para limitar o tempo de execução de um algoritmo é um processo dedutivo. Estudamos uma versão em pseudocódigo do algoritmo, determinamos quais seriam as situações de pior caso para esse algoritmo e usamos ferramentas matemáticas como amortização, somas e equações de recorrências para caracterizar o tempo de execução do algoritmo.

Esta abordagem é poderosa, mas tem suas limitações. A abordagem dedutiva para a análise assintótica nem sempre fornece esclarecimento sobre os fatores constantes que estão "escondidos" na notação O. Da mesma forma, a abordagem dedutiva fornece pouca informação sobre o ponto em que devemos usar um algoritmo assintoticamente lento mas com uma constante pequena, e um algoritmo assintoticamente veloz com uma constante grande. Além disso, a abordagem dedutiva se concentra no pior caso possível, que pode não ser representativo para um dado problema. Finalmente, a abordagem dedutiva falha quando o algoritmo é muito complicado para que limitemos efetivamente seu tempo de execução. Nesses casos, a experimentação pode ajudar-nos a realizar a análise do algoritmo.

Nesta seção, discutiremos algumas técnicas e princípios para realizar a análise experimental de algoritmos.

1.6.1 Escolha do experimento

Ao realizar um experimento, há várias etapas que devem ser percorridas para levá-lo a cabo. Essas etapas requerem tempo e deliberação e devem ser realizadas com cuidado.

Escolhendo a pergunta

A primeira coisa a determinar ao montar um experimento é decidir o que testar. No campo da análise de algoritmos, existem várias possibilidades:

- estimar o tempo de execução assintótico de um algoritmo no caso médio;
- testar entre dois algoritmos qual é mais rápido para uma faixa de valores de entrada $[n_0, n_1]$;

- para um algoritmo que tem parâmetros numéricos, tais como constantes α ou ε que determinam seu comportamento, determinar os valores para esses parâmetros que melhoram o desempenho;
- para um algoritmo que tenta maximizar ou minimizar alguma função de sua entrada, testar quão perto o algoritmo chega do resultado ótimo.

Uma vez que determinamos quais dessas questões (ou uma questão alternativa) deve ser respondida empiricamente, podemos passar para a próxima etapa.

Decidindo o que medir

Quando sabemos qual questão responder, devemos nos concentrar nas medições quantitativas que serão usadas para respondê-la. No caso de um problema de otimização, podemos medir a função que desejamos maximizar ou minimizar. No caso do tempo de execução, o fator a medir não é tão óbvio quanto podemos imaginar.

Podemos, naturalmente, medir o tempo de execução de um algoritmo. Usando um procedimento que retorna o horário da máquina, podemos medir o horário antes e depois de nosso algoritmo ser executado e subtraí-los para determinar quanto tempo a execução usou. Essa medida de tempo é mais útil quando o computador usado é representativo da classe de computadores que desejamos usar para executar nosso algoritmo.

Além disso, devemos reconhecer que o relógio da máquina, usado para medir o tempo de execução da implementação de um algoritmo, pode ser afetado por outros fatores como, por exemplo, outros programas sendo executados concorrentemente, se nosso algoritmo faz uso eficiente de memória *cache* ou se nosso algoritmo usa tanta memória que os dados devem ser escritos em memória auxiliar mais lenta. Todos esses fatores podem retardar um algoritmo essencialmente veloz e, se vamos usar um relógio para medir o tempo de execução, temos de garantir que esses fatores serão minimizados.

Uma abordagem alternativa é medir o tempo de uma forma independente de plataforma, contando o número de vezes que uma operação primitiva é usada em nosso algoritmo. Exemplos dessas operações primitivas que são usadas efetivamente na análise de algoritmos são:

- ***Acessos a memória:*** contando os acessos a memória em um algoritmo de intensa manipulação de dados, obtemos uma medida bastante bem relacionada ao tempo de execução do algoritmo em qualquer máquina.
- ***Comparações:*** em um algoritmo como ordenação, que processa dados realizando comparações sobre pares de elementos, uma contagem das comparações será uma medida bem relacionada ao tempo de execução do algoritmo.
- ***Operações aritméticas:*** em algoritmos numéricos, que são dominados por operações aritméticas, contar o número de adições ou multiplicações pode ser uma maneira efetiva de medir o tempo de execução. Essa medida pode ser transformada em tempo de execução em um dado computador se considerarmos se este computador tem um co-processador aritmético ou não.

Tendo decidido o que desejamos medir, devemos gerar dados de teste para o algoritmo e obter estatísticas.

Gerando dados de teste

Nossas metas ao gerar dados de teste são as seguintes:

- Desejamos gerar amostras para que o cálculo de valores médios forneça resultados estatisticamente significativos.

- Desejamos gerar dados de vários tamanhos para ser capazes de projetar o desempenho do algoritmo em uma faixa de tamanhos de problema.
- Desejamos gerar casos de teste representativos para os dados que serão usados na prática em nosso algoritmo.

Gerar dados que satisfaçam os dois primeiros tópicos é simplesmente uma questão de cobertura, enquanto satisfazer o terceiro critério requer atenção. Precisamos considerar a distribuição de entrada e gerar dados de teste de acordo com essa distribuição. Em geral, produzir dados de teste distribuídos uniformemente não é suficiente. Por exemplo, se nosso algoritmo realiza pesquisa baseado em palavras encontradas em um documento escrito em linguagem natural, então a distribuição das pesquisas não será uniforme. Idealmente, gostaríamos de ter uma forma de coletar dados em quantidade suficiente para obter conclusões estatisticamente válidas. Quando tais dados são disponíveis apenas em parte, podemos gerar dados adicionais que correspondam às propriedades estatísticas dos dados disponíveis. Em qualquer caso, devemos tentar criar dados de teste que permitam-nos tirar conclusões que refutem ou confirmem hipóteses específicas sobre nosso algoritmo.

Codificando a solução e realizando o experimento

A etapa de codificação do nosso algoritmo correta e eficiente exige um certo grau de habilidade em programação. Além disso, se estamos comparando nosso algoritmo a outro, devemos ter certeza de que o outro algoritmo usa o mesmo grau de habilidade em sua codificação que o nosso. O grau de otimização de código da implementação de dois algoritmos que desejamos comparar deve ser tão similar quanto possível. Alcançar essa similaridade é algo subjetivo, mas devemos nos esforçar para obtê-la. Finalmente, devemos nos esforçar por obter resultados que sejam ***reproduzíveis***, ou seja, que um programador diferente com habilidade similar possa reproduzir os resultados através de experimentos similares.

Uma vez que nosso programa esteja completo e que tenhamos gerado os dados de teste, estamos prontos para realizar os experimentos e obter os resultados. Devemos realizar nossos experimentos em um ambiente tão "estéril" quanto possível, eliminando quaisquer fontes de interferência. Devemos também guardar os detalhes desse ambiente, como o número de CPUs, sua velocidade, quantidade de memória disponível, etc.

1.6.2 Análise de dados e visualização

Apresentar dados em tabelas é bastante comum, mas não é tão útil quanto uma representação gráfica. Uma discussão completa destas técnicas de análise de dados e visualização está além do escopo deste livro, mas mesmo assim discutiremos, nesta seção, duas técnicas de análise e visualização usadas para análise de algoritmos.

O teste do quociente

No ***teste do quociente***, usamos conhecimento de nosso algoritmo para obter uma função $f(n) = n^c$ para o termo principal do tempo de execução do algoritmo para alguma constante $c > 0$. Nossa análise é planejada para testar se o tempo de execução médio do algoritmo é $\Theta(n^c)$ ou não. Seja $t(n)$ o tempo de execução de nosso algoritmo em uma instância do problema de tamanho n. O teste do quociente é feito plotando-se o

quociente $r(n) = t(n)/f(n)$, usando vários valores obtidos experimentalmente para $t(n)$ (veja a Figura 1.20).

Se $r(n)$ cresce à medida que n cresce, então $f(n)$ subestima o tempo de execução $t(n)$. Se, por outro lado, $r(n)$ converge para 0, então $f(n)$ superestima o tempo de execução. Por outro lado, se o quociente $r(n)$ converge para uma constante $b > 0$, então teremos encontrado uma boa estimativa para a taxa de crescimento de $t(n)$. Além disso, a constante b fornece uma boa estimativa para o fator constante do tempo de execução $t(n)$.

Ainda assim, devemos reconhecer que testes empíricos podem testar apenas um número finito de dados de entrada e tamanhos de problema; portanto, o teste do quociente não pode ser usado para determinar um valor exato para o expoente $c > 0$. Além disso, sua exatidão é limitada a funções polinomiais para $f(n)$ e, mesmo assim, estudos mostram que o melhor que se pode atingir para determinar o expoente c é limitá-lo ao intervalo $[c - 0{,}5, c + 0{,}5]$.

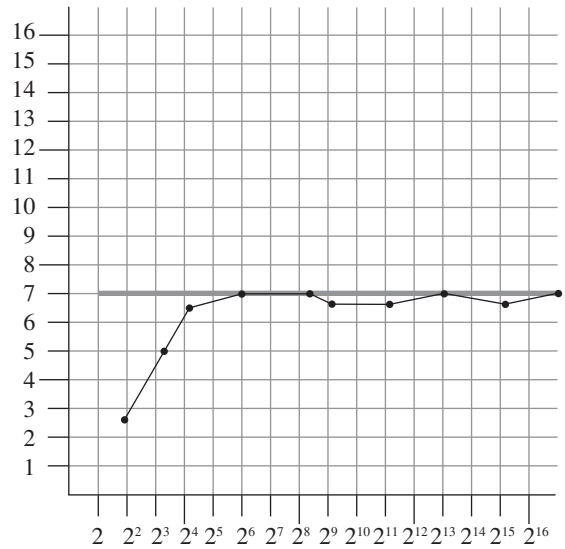


Figura 1.20 Uma plotagem exemplificando o teste do quociente estimando que $r(n) = 7$.

O teste da potência

No ***teste da potência***, podemos obter uma boa estimativa para o tempo de execução $t(n)$ de um algoritmo sem ter uma aproximação inicial. A idéia é usar pares de dados (x,y) obtidos experimentalmente tais que $y = t(x)$, onde x é o tamanho do problema sendo testado, e aplicamos a transformação $(x,y) \rightarrow (x',y')$ onde $x' = \log x$ e $y' = \log y$. Então plotamos os pares (x',y') e examinamos os resultados.

Note que, se $t(n) = bn^c$ para $b > 0$ e $c > 0$, então a transformação implica que $y' = cx' + b$. Assim, se os pares (x',y') estão perto de formar uma linha, então podemos obter os valores das constantes b e c. O expoente c corresponde à inclinação da reta, e o coeficiente b corresponde à altura na qual esta reta encontra o eixo y (veja a Figura 1.21). Alternativamente, se os pares (x',y') crescem de maneira significativa, então podemos deduzir que $t(n)$ é superpolinomial; e se os pares (x',y') convergem para uma constante, é provável que $t(n)$ seja sublinear. Em qualquer caso, por causa dos testes li-

mitados ao tamanho da entrada, é difícil considerar c melhor do que o intervalo $[c - 0{,}5, c + 0{,}5]$ com o teste da potência.

Mesmo assim, o teste do quociente e o teste da potência são geralmente considerados boas abordagens para estimar o tempo de execução empírico de um algoritmo. Eles são consideravelmente melhores do que tentar determinar, através de técnicas de regressão, um polinômio para os dados obtidos nos testes. Essas técnicas geralmente são muito sensíveis a pequenas variações e podem não fornecer boas estimativas para os expoentes em tempos de execução polinomiais.

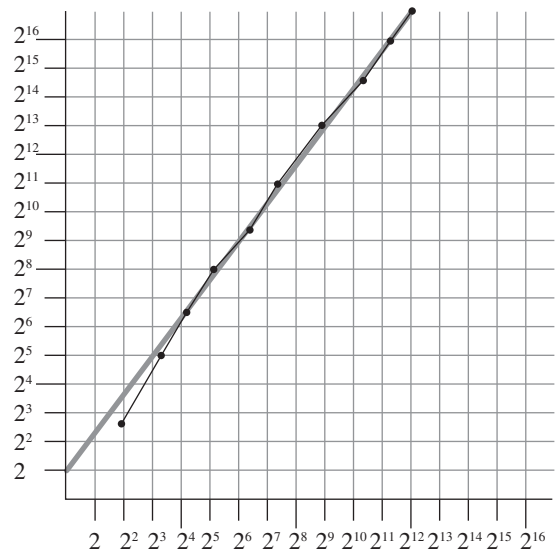


Figura 1.21 Uma plotagem exemplificando o teste da potência. Neste caso, estimamos que $y' = (4/3)x' + 2$; portanto, estimaríamos que $t(n) = 2n^{4/3}$.

1.7 Exercícios

Reforço

R-1.1 Faça um gráfico das funções $12n$, $6n \log(n)$, n^2, n^3 e 2^n usando uma escala logarítmica para os eixos x e y, ou seja, para o valor n e para a função $f(n)$, plote a informação no ponto $(\log(n), \log(f(n)))$.

R-1.2 O algoritmo A usa $10n\log(n)$ operações enquanto o algoritmo B usa n^2 operações. Determine o valor n_0 para o qual A é melhor do que B para $n \geq n_0$.

R-1.3 Repita o problema anterior supondo que B usa $n\sqrt{n}$ operações.

R-1.4 Demonstre que $\log^3(n)$ é $o(n^{1/3})$.

R-1.5 Demonstre que as duas afirmações a seguir são equivalentes:

a. O tempo de execução do algoritmo A é $O(f(n))$.
b. No pior caso, o tempo de execução do algoritmo A é $O(f(n))$.

R-1.6 Ordene a lista de funções a seguir usando a notação O. Agrupe as funções que são Θ uma da outra.

$6n \log n$	2^{100}	$\log \log n$	$\log^2 n$	$2^{\log n}$
2^{2n}	$\lceil \sqrt{n} \rceil$	$n^{0,01}$	$1/n$	$4n^{3/2}$
$3n^{0.5}$	$5n$	$\lfloor 2n \log^2 n \rfloor$	2^n	$n \log_4 n$
4^n	n^3	$n^2 \log n$	$4^{\log n}$	$\sqrt{\log n}$

Dica: quando estiver em dúvida sobre duas funções $f(n)$ e $g(n)$, considere $\log(f(n))$ e $\log(g(n))$ ou $2^{f(n)}$ e $2^{g(n)}$.

R-1.7 Para cada função $f(n)$ e tempo t da tabela a seguir, determine o maior tamanho n de problema que pode ser resolvido em tempo t, supondo que o algoritmo que resolve o problema precisa de $f(n)$ microssegundos. Lembre que $\log(n)$ representa o logaritmo de n em base 2. Alguns elementos já foram preenchidos para ajudá-lo a começar.

	1 Segundo	1 Hora	1 Mês	1 Século
$\log n$	$\approx 10^{300000}$			
$\sqrt{n}$				
n				
$n \log n$				
n^2				
n^3				
2^n				
$n!$		12		

R-1.8 Bill tem um algoritmo find2D para encontrar um elemento x em uma matriz A de tamanho $n \times n$. O algoritmo faz iterações sobre as linhas de A e usa o algoritmo arrayFind do Algoritmo 1.12 em cada linha até que x é encontrado ou todas as linhas foram examinadas. Qual é o tempo de execução de pior caso do algoritmo find2D em função de n? O algoritmo é linear? Justifique.

R-1.9 Considere a equação de recorrência a seguir, definindo $T(n)$ como

$$T(n) = \begin{cases} 4 & \text{se } n = 1 \\ T(n-1) + 4 & \text{se } n > 1, \end{cases}$$

Demonstre, por indução, que $T(n) = 4n$.

R-1.10 Forneça uma estimativa em notação O e em função de n para o método Loop1 dado no Algoritmo 1.22.

R-1.11 Forneça uma análise similar para o método Loop2 dado no Algoritmo 1.22.

R-1.12 Forneça uma análise similar para o método Loop3 dado no Algoritmo 1.22.

R-1.13 Forneça uma análise similar para o método Loop4 dado no Algoritmo 1.22.

R-1.14 Forneça uma análise similar para o método Loop5 dado no Algoritmo 1.22.

```
Algoritmo Loop1(n):
    s ← 0
    para i ← 1 até n faça
      s ← s + i

Algoritmo Loop2(n):
    p ← 1
    para i ← 1 até 2n faça
      p ← p · i

Algoritmo Loop3(n):
    p ← 1
    para i ← 1 até n^2 faça
      p ← p · i

Algoritmo Loop4(n):
    s ← 0
    para i ← 1 até 2n faça
      para j ← 1 até i faça
        s ← s + i

Algoritmo Loop5(n):
    s ← 0
    para i ← 1 até n^2 faça
      para j ← 1 até i faça
        s ← s + i
```

Algoritmo 1.22 Uma coleção de laços.

R-1.15 Demonstre que, se $f(n)$ é $O(g(n))$ e $d(n)$ é $O(h(n))$, então $f(n) + d(n)$ é $O(g(n) + h(n))$.

R-1.16 Demonstre que $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$.

R-1.17 Demonstre que $f(n)$ é $O(g(n))$ se e somente se $g(n)$ é $\Omega(f(n))$.

R-1.18 Demonstre que, se $p(n)$ é um polinômio, então, $\log(p(n))$ é $O(\log(n))$.

R-1.19 Demonstre que $(n + 1)^5$ é $O(n^5)$.

R-1.20 Demonstre que 2^{n+1} é $O(2^n)$.

R-1.21 Demonstre que n é $o(n\log(n))$.

R-1.22 Demonstre que n^2 é $\omega(n)$.

R-1.23 Demonstre que $n^3\log(n)$ é $\Omega(n^3)$.

R-1.24 Demonstre que $\lceil f(n) \rceil$ é $O(f(n))$ se $f(n)$ é uma função positiva não decrescente sempre maior do que 1.

R-1.25 Justifique o fato de que, se $d(n)$ é $O(f(n))$ e $e(n)$ é $O(g(n))$, então $d(n)e(n)$ é $O(f(n)g(n))$.

R-1.26 Qual o tempo de execução amortizado de uma operação em uma série de n operações de add em uma tabela expansível inicialmente vazia implementada com um arranjo de forma que o parâmetro capacityIncrement é sempre

$\lceil \log(m + 1) \rceil$, onde m é o número de elementos na tabela? Ou seja, a tabela é sempre expandida em $\lceil \log(m + 1) \rceil$ células, e capacityIncrement passa a valer $\lceil \log(m' + 1) \rceil$, onde m é o tamanho antigo da tabela e m' é o novo tamanho (em termos do número de elementos presentes).

R-1.27 Descreva um algoritmo recursivo para determinar o menor e o maior valor em um arranjo A com n elementos. Seu método deve retornar um par (a,b) onde a é o menor elemento e b, o maior. Qual o tempo de execução de seu método?

R-1.28 Reescreva a prova do Teorema 1.31 sob a hipótese de que o custo de aumentar um arranjo do tamanho k para o tamanho $2k$ é $3k$ ciberdólares. Quanto deve custar cada operação add para fazer funcionar a amortização?

R-1.29 Use o teste do quociente para plotar e comparar o conjunto de pontos abaixo:

$$S = \{(1,1), (2,7), (4,30), (8,125), (16,510), (32,2045), (64,8190)\}$$

com as seguintes funções:

a. $f(n) = n$
b. $f(n) = n^2$
c. $f(n) = n^3$.

R-1.30 Use o teste da potência para estimar uma função polinomial $f(n) = bn^c$ que corresponda aos dados abaixo, plotando-os.

$$S = \{(1,1), (2,7), (4,30), (8,125), (16,510), (32,2045), (64,8190)\}.$$

Criatividade

C-1.1 Qual o tempo de execução amortizado das operações em uma seqüência de n operações $P = p_1p_2 \ldots p_n$ se o tempo de execução de p_i é $\Theta(i)$ se i é múltiplo de 3 e constante em caso contrário?

C-1.2 Seja $P = p_1p_2 \ldots p_n$ uma seqüência de n operações, cada uma sendo uma operação red ou blue, com p_1 sendo red e p_2 sendo blue. O tempo de execução de uma operação blue é constante. O tempo de execução da primeira operação red é constante, mas cada operação red a seguir leva o dobro do tempo da operação red anterior. Qual o tempo de execução amortizado das operações red e blue sob as hipóteses abaixo?

a. Existem sempre $\Theta(1)$ operações blue entre operações red consecutivas.
b. Existem sempre $\Theta(\sqrt{n})$ operações blue entre operações red consecutivas.
c. O número de operações blue entre uma operação red p_i e a operação red anterior p_j é sempre duas vezes o número entre p_j e a sua operação red anterior.

C-1.3 Qual o tempo de execução total para contar de 1 a n em binário se o tempo necessário para somar 1 ao número i é proporcional ao número de *bits* na representação binária de i que devem ser alterados para representar $i + 1$?

C-1.4 Considere a equação de recorrência a seguir, definindo $T(n)$:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(n-1) + n & \text{se } n > 1 \end{cases}$$

Demonstre por indução que $T(n) = n(n + 1)/2$.

C-1.5 Considere a equação de recorrência a seguir, definindo $T(n)$:

$$T(n) = \begin{cases} 1 & \text{se } n = 0 \\ T(n-1) + 2^n & \text{do contrário} \end{cases}$$

Demonstre por indução que $T(n) = 2^{n+1} - 1$.

C-1.6 Considere a equação de recorrência a seguir, definindo $T(n)$:

$$T(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2T(n-1) & \text{do contrário} \end{cases}$$

Demonstre por indução que $T(n) = 2^n$.

C-1.7 Al e Bill estão discutindo sobre a performance de seus algoritmos de ordenação. Al afirma que seu algoritmo executado em tempo $O(n \log(n))$ é sempre superior ao algoritmo de Bill, que é $O(n^2)$. Para decidir, eles implementam e executam os dois algoritmos em uma série de dados de teste. Para o espanto de Al, eles descobrem que quando $n < 100$ o algoritmo $O(n^2)$ é mais rápido e apenas quando $n \geq 100$ é que o algoritmo $O(n \log(n))$ é melhor. Explique como esta situação pode ocorrer. Se for preciso, forneça exemplos numéricos.

C-1.8 A segurança de comunicações é extremamente importante em redes de computadores, e uma forma típica de garantir essa segurança é criptografar as mensagens. *Sistemas criptográficos* típicos para a transmissão segura de dados através de redes são baseados no fato de que não são conhecidos algoritmos eficientes para a fatoração de números inteiros muito grandes. Assim, se podemos representar uma mensagem secreta como um grande número primo p, podemos transmitir pela rede o número $r = p \cdot q$, onde $q > p$ é outro grande número primo que serve como *chave de encriptação*. Um espião que obtenha o número r transmitido pela rede teria de fatorá-lo para descobrir a mensagem secreta p.

Usar fatoração para descobrir a mensagem é bastante difícil sem que se conheça a chave q. Para entender por que, considere o seguinte algoritmo simples de fatoração:

> Para cada inteiro p tal que $1 < p < r$, verifique se p divide r. Se dividir, imprima a mensagem "A mensagem secreta é p"; em caso contrário, continue.

a. Supondo que o espião use esse algoritmo em um computador que possa realizar em 1 microssegundo (ou seja, em um milionésimo de segundo) uma operação de divisão entre dois inteiros de 100 *bits* cada um. Forneça uma estimativa do tempo necessário (no pior caso) para decifrar a mensagem secreta se r tem 100 *bits*.
b. Qual a complexidade de pior caso do algoritmo acima? Já que a entrada do algoritmo é um grande número r, suponha que o tamanho n da entrada é o número de *bytes* necessário para armazenar r, ou seja, $n = (\log_2(r))/8$, e que cada divisão toma tempo proporcional a $O(n)$.

C-1.9 Dê um exemplo de uma função positiva $f(n)$ de tal forma que $f(n)$ não seja nem $O(n)$ nem $\Omega(n)$.

C-1.10 Demonstre que $\sum_{i=1}^{n} i^2$ é $O(n^3)$.

C-1.11 Demonstre que $\sum_{i=1}^{n} i/2^i < 2$.
Dica: tente limitar esta soma, termo a termo, com uma progressão geométrica.

C-1.12 Demonstre que $\log_b(f(n))$ é $\Theta(\log(f(n)))$, se $b > 1$ for uma constante.

C-1.13 Descreva uma maneira de encontrar o mínimo e o máximo de n números usando menos de $3n/2$ comparações.
Dica: primeiramente construa um grupo de candidatos a mínimo e um grupo de candidatos a máximo.

C-1.14 Suponha que você recebeu um conjunto de pequenas caixas exatamente iguais, numeradas de 1 a n. As primeiras i caixas contêm uma pérola, enquanto as outras $n - i$ estão vazias. Você também tem duas varinhas mágicas que podem testar se uma caixa está vazia ou não, mas cada varinha desaparecerá se for testada em uma caixa vazia. Mostre que, sem conhecer o valor de i, você pode usar as varinhas para determinar todas as caixas contendo pérolas usando no máximo $o(n)$ testes. Expresse, como função de n, o número assintótico de testes necessários.

C-1.15 Repita o problema acima supondo que você tem k varinhas mágicas, com $k > 2$ e $k < \log n$. Expresse, como função de n e k, o número assintótico de testes necessários para identificar as caixas com as pérolas.

C-1.16 Um ***polinômio*** p de grau n é uma equação da forma

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

onde x é um número real e cada a_i é uma constante.

a. Descreva um método $O(n^2)$ para avaliar $p(x)$ para um valor particular de x.
b. Considere que reescrevemos $p(x)$ como

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n) \cdots))),$$

que é conhecido como ***método de Horner***. Caracterize, usando a notação O, o número de multiplicações e adições que esse método usa.

C-1.17 Considere a "justificativa" abaixo, mostrando que todas as ovelhas de um rebanho são da mesma cor:
Caso base: uma ovelha. Com certeza, ela é de sua própria cor.
Passo da indução: suponha um rebanho de n ovelhas. Tome uma ovelha a do rebanho. As outras $n - 1$ são da mesma cor, por indução. Agora coloque a ovelha a de volta no rebanho e retire uma ovelha diferente b. Por indução, as $n - 1$ ovelhas (agora com a de volta ao rebanho) são todas da mesma cor. Então a é da mesma cor das outras ovelhas e, portanto, todas as ovelhas são da mesma cor.
O que está errado nessa "justificativa"?

C-1.18 Considere a "justificativa" abaixo, mostrando que a função de Fibonacci $F(n)$, definida como $F(1) = 1$, $F(2) = 2$, $F(n) = F(n - 1) + F(n - 2)$, é $O(n)$:
Caso base ($n \leq 2$): $F(1) = 1$, que é $O(1)$ e $F(2) = 2$, que é $O(2)$.
Passo da indução ($n > 2$): assuma que a afirmação é verdadeira para $n' < n$. Considere n e o fato de que $F(n) = F(n - 1) + F(n - 2)$. Por indução $F(n - 1)$ é $O(n - 1)$ e $F(n - 2)$ é $O(n - 2)$. Então $F(n)$ é $O((n - 1) + (n - 2))$ pela identidade apresentada no exercício R-1.15. Portanto, $F(n)$ é $O(n)$ já que $O((n - 1) + (n - 2))$ é $O(n)$.
O que está errado nessa "prova"?

C-1.19 Considere a função de Fibonacci $F(n)$ definida no exercício anterior. Demonstre, por indução, que $F(n)$ é $\Omega((3/2)^n)$.

C-1.20 Justifique o Teorema 1.13 através de uma ilustração análoga à Figura 1.11b para quando n é ímpar.

C-1.21 Um arranjo A contém $n - 1$ inteiros não repetidos com valores entre 0 e $n - 1$, ou seja, existe um número desta faixa de valores que não está em A. Apresente um algoritmo com tempo de execução $O(n)$ para encontrar este número. É permitido que você use espaço adicional que seja $O(1)$ além do próprio arranjo A.

C-1.22 Demonstre que $\sum_{i=1}^{n} \lceil \log_2(i) \rceil$ é $O(n\log(n))$.

C-1.23 Demonstre que $\sum_{i=1}^{n} \lceil \log_2(i) \rceil$ é $\Omega(n\log(n))$.

C-1.24 Demonstre que $\sum_{i=1}^{n} \lceil \log_2(n/i) \rceil$ é $O(n)$. Você pode assumir que n é uma potência de 2.
Dica: use indução para reduzir o problema de n a $n/2$.

C-1.25 Um rei cruel tem uma adega que contém n garrafas de vinhos raros, e seus guardas acabam de capturar um espião tentando envenenar o vinho do rei. Felizmente os guardas capturaram-no depois que ele havia envenenado apenas uma das garrafas, mas, infelizmente, ninguém sabe qual. Para piorar tudo, o veneno é muito poderoso, mesmo a menor parte dele ainda pode matar uma pessoa. Ainda assim, o veneno age devagar: leva um mês exato para que uma pessoa morra. Ache uma maneira que permita ao rei determinar em um mês qual a garrafa que contém o veneno, usando no máximo $O(\log(n))$ dos seus provadores de alimento.

C-1.26 Seja S um conjunto de n linhas tais que não há duas linhas paralelas e não existem três linhas que se cruzem no mesmo ponto. Demonstre, por indução, que as linhas em S determinam $\Theta(n^2)$ pontos de intersecção.

C-1.27 Suponha que cada linha de uma matriz $A_{n \times n}$ consiste de valores 1 e 0 de tal forma que em cada linha de A todos os valores 1 vêm antes dos valores 0. Descreva um método que tem tempo de execução $O(n)$ (e não $O(n^2)$) para encontrar a linha de A que contém o maior número de valores 1.

C-1.28 Suponha que cada linha de uma matriz $A_{n \times n}$ consiste de valores 1 e 0 de tal forma que em cada linha i de A todos os valores 1 vêm antes dos valores 0. Suponha, além disso, que na linha i o número de valores 1 é no mínimo o número de valores 1 na linha $i + 1$ para $i = 0, 1, ..., n - 2$. Descreva um método que tem tempo de execução $O(n)$ (e não $O(n^2)$) para contar o número de valores 1 em A.

C-1.29 Descreva em pseudocódigo um método para multiplicar uma matriz $A_{n \times m}$ e uma matriz $B_{m \times p}$. Lembre que o produto $C = AB$ é definido de tal forma que $C[i][j] = \sum_{k=1}^{m} A[i][k] \cdot B[k][j]$. Qual o tempo de execução de seu algoritmo?

C-1.30 Apresente um algoritmo recursivo para calcular o produto de dois inteiros positivos m e n usando apenas adição.

C-1.31 Forneça o pseudocódigo para uma nova classe ShrinkingTable que realiza o método add da tabela extensível bem como os métodos remove( ), que removem o último elemento da tabela e shrinkToFit( ), que substitui o arranjo implementando a tabela com um arranjo cujo tamanho é exatamente igual ao número de elementos na tabela.

C-1.32 Considere uma tabela extensível que suporte os métodos add e remove como definidos no exercício anterior. Suponha que o arranjo implementando a tabela cresce dobrando sua capacidade quando precisa ser aumentado, e é diminuído pela metade quando o número de elementos em uso é menor do que $N/4$, onde N é o tamanho atual do arranjo. Demonstre que uma seqüência de n operações de add e remove começando com um arranjo com $N = 1$ toma tempo $O(n)$.

C-1.33 Considere uma tabela extensível que, em vez de copiar os elementos da tabela em um arranjo com o dobro do tamanho (ou seja, de N para $2N$), copia os elementos para um arranjo com $\lceil \sqrt{N} \rceil$ posições adicionais, passando de N para $N + \lceil \sqrt{N} \rceil$. Demonstre que realizar uma seqüência de n operações add (ou seja, inserções ao final da tabela) pode ser feito em tempo $\Theta(n^{3/2})$.

Projetos

P-1.1 Programe os algoritmos prefixAverages1 e prefixAverages2 da Seção 1.4 e faça uma análise experimental dos seus tempos de execução. Trace seus tempos de execução em um gráfico em função do tamanho das entradas, tanto em escala linear quanto em escala logarítmica. Escolha valores representativos para n e rode pelo menos cinco testes para cada valor de n escolhido.

P-1.2 Realize uma análise experimental que compare os tempos de execução relativos dos métodos mostrados no Algoritmo 1.22. Use tanto o teste de quociente como o teste de potência para estimar os tempos de execução dos vários métodos.

P-1.3 Implemente uma tabela expansível usando arranjos que possam aumentar de tamanho à medida que elementos são adicionados. Realize uma análise experimental dos tempos de execução para efetuar uma seqüência de operações add, supondo que o tamanho do arranjo é aumentado de N para cada um dos possíveis valores:

a. $2N$
b. $N + \lceil \sqrt{N} \rceil$
c. $N + \lceil \log N \rceil$
d. $N + 100$.

Notas

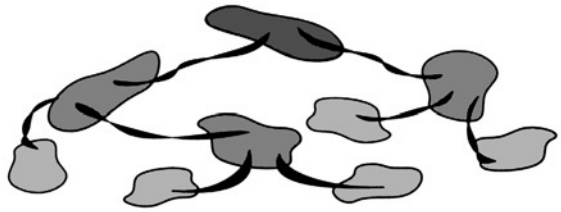
Os tópicos discutidos neste capítulo provêm de várias origens. A amortização tem sido usada para analisar uma variedade de estruturas de dados e algoritmos, mas não tornou-se um tema de estudo mais profundo até a metade da década de 1980. Para mais informação sobre amortização, veja o artigo de Tarjan [201] ou o livro de Tarjan [200].

Nosso uso da notação O é coerente com o uso da maioria dos autores, mesmo que tenhamos usado uma abordagem mais conservadora do que alguns deles. A notação O iniciou várias discussões na comunidade de análise de algoritmos e teoria da computação, muitas delas a respeito de seu uso adequado [37, 92, 120]. Knuth [118, 120], por exemplo, a define usando a notação $f(n) = O(g(n))$, mas refere-se a esta "igualdade" como sendo "em mão única", mesmo mencionando o fato de que a notação O define, na verdade, um conjunto de funções. Preferimos adotar uma visão mais convencional da igualdade e ver O como um conjunto, seguindo a sugestão de Brassard [37]. O leitor interessado em estudar a análise de caso médio pode consultar o capítulo escrito por Vitter e Flajolet [207].

Incluímos uma série de fatos matemáticos úteis no Apêndice A. O leitor interessado em mais detalhes da análise de algoritmos pode consultar o livro de Graham, Knuth e Patashnik [90] ou o de Sedgewick e Flajolet [184]. O leitor interessado na história da matemática pode consultar o livro de Boyer e Merzbach [35]. Nossa versão da história de Arquimedes foi retirada de [155]. Finalmente, para mais informação sobre o uso de experimentação para estimar o tempo de execução de algoritmos, referenciamos ao leitor vários artigos de McGeoch e seus co-autores [142, 143, 144].

Capítulo

2 Estruturas de dados básicas



2.1 Pilhas e filas 70
2.1.1 Pilhas 70
2.1.2 Filas` 73
2.2 Vetores, listas e seqüências 77
2.2.1 Vetores 77
2.2.2 Listas 80
2.2.3 Seqüências 84
2.3 Árvores 86
2.3.1 O tipo abstrato de dados árvore 89
2.3.2 Caminhamento em árvores 90
2.3.3 Árvores binárias 95
2.3.4 Estruturas de dados para representar árvores 100
2.4 Filas de prioridade e *heaps* 104
2.4.1 O tipo abstrato de dados fila de prioridade 104
2.4.2 *PQ-Sort, selection-sort* e *insertion-sort* 106
2.4.3 A estrutura de dados *heap* 108
2.4.4 *Heapsort* 116
2.5 Dicionários e tabelas hash 122
2.5.1 O TAD dicionário não-ordenado 122
2.5.2 Tabelas *hash* 124
2.5.3 Funções *hash* 125
2.5.4 Mapas de compressão 127
2.5.5 Métodos de tratamento de colisões 128
2.5.6 *Hash* universal 132
2.6 Exemplo em Java: *heap* 135
2.7 Exercícios 138

Estruturas de dados básicas, como pilhas e filas, são usadas em uma gama variada de aplicações. O uso de boas estruturas de dados é, em geral, a diferença entre um algoritmo eficiente e um ineficiente. Desta forma, percebemos a importância de revisar e discutir diferentes estruturas de dados básicas.

Iniciaremos a discussão deste capítulo pelo estudo de pilhas e filas, incluindo a forma pela qual são usadas para implementar recursão e multiprogramação. Continuaremos a discussão introduzindo os TADs ***vector****, ***list***** e ***sequence******, que representam cada qual uma coleção de elementos organizados linearmente e oferecem métodos para acessar, inserir e remover elementos arbitrários. Uma propriedade importante das seqüências é o fato de que, da mesma forma que pilhas e filas, a ordem dos elementos é determinada pelas operações especificadas no tipo abstrato de dados, e não pelos valores dos elementos.

Além destas estruturas de dados lineares, discutiremos também estruturas não-lineares, que se utilizam de relações organizacionais mais ricas do que simplesmente "antes" e "depois". Em especial, discutiremos o tipo abstrato de dados ***tree*******, que define relacionamentos hierárquicos, com alguns objetos estando "acima" e alguns "abaixo" de outros. A terminologia usada com árvores é originária de árvores genealógicas, de forma que os termos "pai", "filho", "ancestral" e "descendente" são palavras normalmente empregadas para descrever esses relacionamentos.

Neste capítulo, também estudaremos estruturas de dados que armazenam "elementos com prioridades", isto é, elementos com prioridades associadas aos mesmos. Tais prioridades são normalmente representadas por valores numéricos, mas podemos entendê-las como objetos arbitrários, desde que sejam uma forma consistente de comparar pares de objetos. Uma fila com prioridade nos permite selecionar e remover o elemento com a primeira prioridade, que determinamos, sem perda de generalidade, que seja o elemento com a menor chave. Esta visão geral nos permite definir um tipo abstrato de dados genérico chamado ***priority queue***, para armazenar e recuperar elementos com prioridade. Esse TAD é de natureza distinta das estruturas de dados baseadas em posição discutidas neste capítulo, tais como pilhas, filas, seqüências e mesmo árvores. Essas estruturas de dados armazenam elementos em posições específicas que correspondem a posições em um arranjo linear de elementos determinado pelas operações de inserção e remoção efetuadas. O TAD fila com prioridades armazena os elementos de acordo com suas prioridades, e não possui a noção de "posição".

A última estrutura que apresentaremos é o dicionário, que armazena elementos de forma que eles possam ser localizados rapidamente usando chaves. A motivação para tais pesquisas é que cada elemento em um dicionário normalmente armazena informação adicional útil por trás de sua chave de busca, mas a única maneira de ter acesso a esta informação é usando a chave de busca. Contudo, uma relação de ordem total sobre as chaves é sempre exigida para uma fila com prioridades; isso é opcional para um dicionário. Da mesma forma, a forma mais simples de dicionário, que usa uma tabela *hash*, presume apenas que podemos atribuir um inteiro para cada chave e determina quando duas chaves são iguais.

* N. de T. Vetor.
** N. de T. Lista.
*** N. de T. Seqüência.
**** N. de T. Árvore.

2.1 Pilhas e filas

2.1.1 Pilhas

Uma ***pilha*** é um contêiner de objetos que são inseridos e removidos de acordo com o princípio ***o último que entra é o primeiro que sai*** (*LIFO**). Objetos podem ser inseridos em uma pilha a qualquer momento, mas apenas o que foi inserido mais recentemente (isto é, "o último") pode ser removido a qualquer momento. O nome "pilha" é derivado da metáfora da pilha de pratos de um dispositivo automático de entrega de pratos de uma lanchonete. Neste caso, as operações fundamentais envolvem "empilhar"** e "desempilhar"*** pratos da pilha.

Exemplo 2.1: *Navegadores para Internet armazenam os endereços dos locais recentemente visitados em uma pilha. Cada vez que o usuário visita uma nova página, seu endereço é "empilhado" na pilha de endereços. A partir de então, o navegador permite que o usuário "desempilhe" os endereços usando o botão "voltar", retornando a páginas previamente visitadas.*

O tipo abstrato de dados pilha

Uma pilha S é um tipo abstrato de dados (TAD) que suporta os seguintes métodos:

push(o): insere o objeto o no topo da pilha.

pop(): remove o objeto do topo da pilha retornando-o, ou seja, retorna o elemento mais recentemente inserido que ainda se encontra na pilha; um erro ocorre se a pilha estiver vazia.

Adicionalmente, podemos definir os seguintes métodos:

size(): retorna o número de objetos na pilha.

isEmpty(): retorna um booleano que indica se a pilha está vazia.

top(): retorna o objeto do topo da pilha sem removê-lo da mesma; um erro ocorre se a pilha estiver vazia.

Uma implementação simples baseada em arranjo

Uma pilha é facilmente implementada usando-se um arranjo S de N elementos, em que os elementos são armazenados de $S[0]$ a $S[t]$, onde t é o inteiro que indica o índice do elemento que esta no topo em S. Observe que um dos detalhes mais importantes de tal implementação é que devemos especificar um tamanho máximo N para nossa pilha, digamos, $N = 1000$ (veja a Figura 2.1).

* N. de T. Em inglês *Last-In First-Out*.
** N. de T. Em inglês *push*.
*** N. de T. Em inglês *pop*.

Figura 2.1 Implementando uma pilha com um arranjo S. O elemento do topo está na célula $S[t]$.

Lembrando que a convenção deste livro é que os arranjos iniciam no índice 0, inicializamos t com –1 e usamos este valor para determinar quando a pilha está vazia. Da mesma forma, podemos utilizar essa variável para determinar o número de elementos na pilha (t + 1). Podemos também sinalizar uma situação de erro se tentarmos inserir um elemento novo quando o arranjo S estiver cheio. Considerando esta nova exceção, podemos implementar os métodos principais do TAD pilha como descrito no Algoritmo 2.2.

Algoritmo push(o):
  se size() = N **então**
    indica que um erro de pilha cheia ocorreu
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

Algoritmo pop():
  se isEmpty() **então**
    indica que um erro de pilha vazia ocorreu
  $e \leftarrow S[t]$
  $S[t] \leftarrow$ **null**
  $t \leftarrow t - 1$
  retorna e

Algoritmo 2.2 Implementação de uma pilha usando um arranjo

Descrições em pseudocódigo de métodos para o TAD pilha que executem em tempo constante são diretas. Cada um dos métodos da pilha na implementação usando arranjo executa uma certa quantidade de sentenças que envolvem expressões aritméticas, comparações e atribuições. Ou seja, nesta implementação do TAD pilha, cada método executa em tempo $O(1)$.

A implementação usando arranjo de uma pilha é ao mesmo tempo simples e eficiente, e é amplamente utilizada em uma grande variedade de aplicações computacionais. Apesar disso, esta implementação tem um aspecto negativo; ela precisa assumir um limite superior fixo N para o tamanho máximo da pilha. Uma aplicação pode, na verdade, necessitar menos espaço que isso e, nesse caso, estamos desperdiçando memória. Por outro lado, uma aplicação pode necessitar mais espaço que isso, caso em que nossa implementação de pilha pode "derrubar" a aplicação com um erro tão-logo tente-se inserir o objeto (N + 1) na pilha. Desta forma, mesmo sendo simples e eficiente, a implementação baseada em arranjo não é necessariamente a ideal. Felizmente, existem outras implementações, discutidas mais adiante neste capítulo, que não tem limitação de tamanho e ocupam espaço proporcional ao número de elementos armazenados na pilha. Outra opção é usar uma tabela estendida, como discutido na Seção 1.5.2. Entretanto, nos casos onde houver uma boa estimativa do número de itens que se necessita armazenar na pilha, a implementação baseada em arranjo é de longe a melhor. As pilhas cumprem um papel vital em uma grande quantidade de aplicações com-

putacionais, de maneira que é extremamente útil dispor de uma implementação rápida deste TAD, tal como a implementação simples usando arranjo.

Usando pilhas em chamadas de funções e recursão

As pilhas têm um papel importante nos ambientes de execução das linguagens procedurais modernas, tais como C, C++ e Java. Cada *thread* em um programa em execução escrito em uma destas linguagens possui uma pilha privada, chamada de ***pilha de métodos***, que é usada para manter as variáveis locais e outras importantes informações sobre os métodos à medida que são invocados durante a execução (veja a Figura 2.3).

Mais especificamente, durante a execução da *thread* do programa, o ambiente de execução mantém uma pilha cujos elementos são descritores das invocações dos métodos ativos no momento (isto é, não terminados). Esses descritores são chamados de ***frames***. O *frame* para a invocação do método "cool" armazena os valores correntes das variáveis locais e os parâmetros deste método, bem como informações sobre o método que o ativou e sobre o que este método retorna.

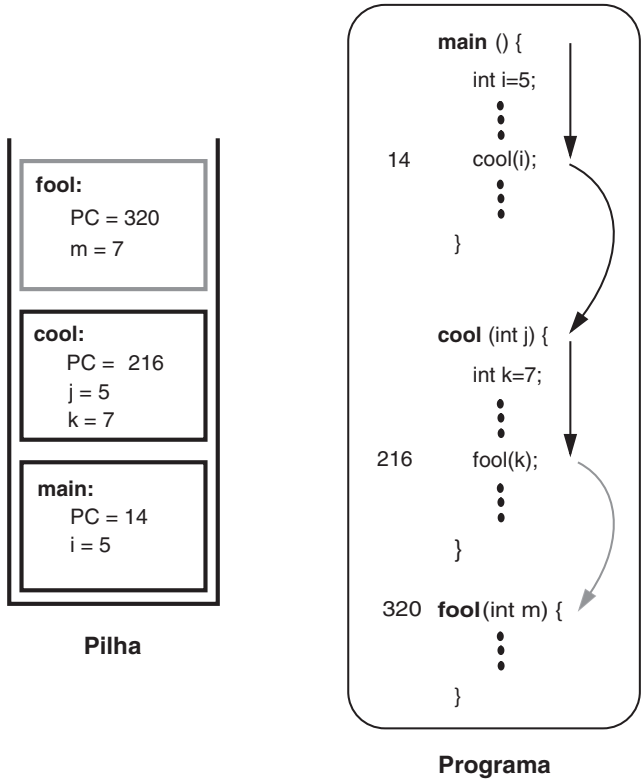


Figura 2.3 Exemplo de pilha de métodos: o método fool recém foi chamado pelo método cool, o qual, por sua vez, fora ativado anteriormente pelo método main. Observe os valores do contador de instruções, parâmetros e variáveis locais armazenados nos *frames* da pilha. Quando a invocação do método fool terminar, a invocação do método cool irá reiniciar sua execução na instrução 217, o que é determinado incrementando-se o valor do contador de programa armazenado no *frame* da pilha.

O ambiente de execução mantém o endereço da instrução que a *thread* está executando em determinado instante no programa em um registrador especial chamado ***contador de instruções***. Quando o método "cool" invoca outro método "fool", o valor corrente do contador de instruções é gravado no *frame* correspondente à invocação corrente de cool (de maneira que o computador "saiba" para onde retornar quando o método fool tiver encerrado).

No topo da pilha de métodos, localiza-se o *frame* do ***método em execução***, ou seja, o método que possui o controle da execução no momento. Os demais elementos da pilha são *frames* dos ***métodos suspensos***, isto é, métodos que foram invocados por outros e, no momento, aguardam o retorno do controle até seu encerramento. A ordem dos elementos na pilha corresponde à cadeia de ativações dos métodos correntemente ativos. Quando um novo método é invocado, um *frame* para este método é inserido na pilha (*push*). Quando ele se encerra, seu *frame* é retirado da pilha (*pop*), e o computador reinicia a execução do método que havia sido suspenso anteriormente.

A pilha de métodos também executa a passagem de parâmetros para os métodos. Muitas linguagens, tais como C e Java, usam o protocolo de passagem de parâmetro ***por valor*** através desta pilha. Isso significa que o ***valor*** atual de uma variável (ou expressão) é o que é passado como argumento na chamada de um método. No caso de uma variável x de um tipo primitivo, tal como int ou float, o valor corrente de x é simplesmente o número associado com x. Quando esse valor é passado para o método chamado, ele é atribuído para uma variável local localizada no *frame* do método acionado (esta atribuição simples também pode ser vista na Figura 2.3). Observe que, se o método chamado alterar o valor desta variável local, isso ***não*** irá alterar o valor da variável no método chamador.

Recursão

Uma das vantagens de se usar uma pilha para implementar a invocação dos métodos é que isso possibilita aos programas usar ***recursão*** (Seção 1.1.4). Isto é, permite a um método ativar a si mesmo como uma sub-rotina.

Lembramos que, usando essa técnica corretamente, devemos sempre projetar métodos recursivos que tenham a garantia de encerrar em algum ponto (por exemplo, fazendo chamadas recursivas para instâncias "menores" do problema e tratando algumas destas instâncias "menores" de forma não-recursiva, como casos especiais). Observamos que, se projetarmos um método de "recursão infinita", ele não irá executar para sempre. Ao invés disso, em algum momento, toda a memória disponível para a pilha de métodos será usada, e um erro de falta de memória será gerado. Se usarmos recursão com cuidado, entretanto, a pilha de métodos irá implementar métodos recursivos sem nenhum problema. Cada chamada do mesmo método será associada a um *frame* distinto, completo, com seus próprios valores para as variáveis locais. A recursão pode ser muito poderosa, na medida em que nos permite projetar programas simples e eficientes para uma gama de problemas difíceis.

2.1.2 Filas

Outra estrutura de dados básica é a ***fila***. Prima próxima da pilha, também consiste de um contêiner de objetos que são inseridos e removidos de acordo com o princípio ***o primeiro que entra é o primeiro que sai*** (***FIFO****). Ou seja, os elementos podem ser

* N. de T. Em inglês *First-In First-Out*.

inseridos a qualquer momento, mas apenas o elemento que está a mais tempo na fila pode ser removido. Normalmente dizemos que os elementos entram na fila pelo ***fim*** e são removidos pela ***frente***.

O tipo abstrato de dados fila

O tipo abstrato de dados fila mantém os objetos em uma seqüência onde o acesso aos elementos e a remoção dos mesmos são restritos ao primeiro elemento da seqüência, que é identificado como elemento da ***frente*** da fila, enquanto que a inserção é restrita ao final da seqüência, que é chamada de ***fim*** da fila. Desta forma, reforçamos a regra que determina que os elementos são inseridos e removidos de acordo com o princípio FIFO. O TAD ***fila*** suporta os dois métodos fundamentais a seguir:

enqueue(o): insere o objeto o no fim da fila.

dequeue(): remove e retorna o elemento da frente da fila; um erro ocorre se a fila estiver vazia.

Além disso, o TAD fila inclui suporte para os métodos que seguem:

size(): retorna o número de objetos na fila.

isEmpty(): retorna um valor booleano que indica quando a fila está vazia.

front(): retorna mas não remove o objeto da frente da fila; um erro ocorre se a fila estiver vazia.

Uma implementação simples usando arranjo

Apresentamos uma implementação simples de uma fila usando um arranjo Q de capacidade N para armazenar seus elementos. Considerando que a regra principal do TAD fila diz que os elementos devem ser inseridos e retirados de acordo com o princípio FIFO, podemos decidir como iremos nos manter informados sobre a frente e o fim da fila.

Para evitar movimentar os objetos uma vez que estejam colocados em Q, definiremos duas variáveis, f e r, que têm o seguinte significado:

- f é um índice para a célula de Q que armazena o primeiro elemento da fila (que é o próximo candidato a ser removido por uma operação dequeue), a menos que a fila esteja vazia (caso em que $f = r$).
- r é um índice da próxima célula livre em Q.

Inicialmente, atribuímos $f = r = 0$, e indicamos que a fila está vazia pela condição $f = r$. A partir de então, quando quisermos remover um elemento da frente da fila, simplesmente incrementamos f para o índice da próxima célula. Da mesma forma, quando acrescentamos um elemento, podemos simplesmente incrementar r para o índice da próxima célula livre de Q. Temos de ser um pouco cuidadosos, entretanto, para não estourar o fim do arranjo. Considere o que acontece, por exemplo, se repetidamente enfileirarmos e desenfileirarmos um único elemento N vezes. Teremos $f = r = N$. Se tentarmos inserir o elemento apenas mais uma vez, iremos obter um erro de índice-fora-de-faixa (uma vez que as N posições válidas de Q vão de $Q[0]$ a $Q[N-1]$), mesmo que haja espaço sobrando na fila neste caso. Para evitar esse problema e permitir que se usem todas as células do arranjo Q, deixamos os índices f e r "girarem ao redor" do fim de Q. Isto é, enxergamos Q como um "arranjo circular" que vai de $Q[0]$ até $Q[N-1]$ e então retorna imediatamente para $Q[0]$ (veja a Figura 2.4).

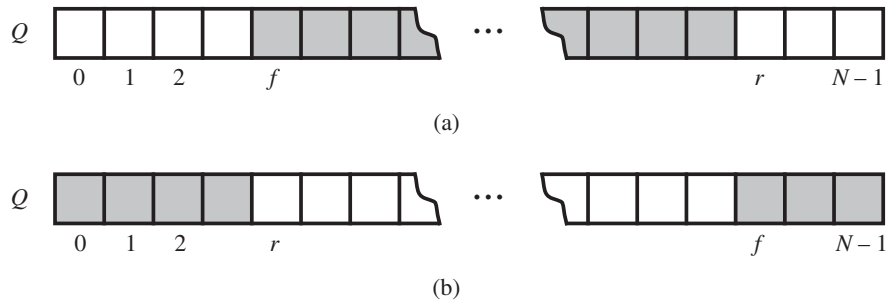


Figura 2.4 Usando o arranjo Q de forma circular: (a) a configuração "normal" com $f \leq r$; (b) a configuração "que gira" com $r < f$. As células que armazenam os elementos estão destacadas.

Implementar essa visão circular de Q é muito fácil. Cada vez que incrementamos f ou r, simplesmente calculamos seu incremente como "$(f + 1) \bmod N$" ou "$(r + 1) \bmod N$", respectivamente, onde "mod" é o operador ***módulo***, que é calculado tomando-se o resto da divisão inteira, de maneira que, se y é diferente de zero, então

$$x \bmod y = x - \lfloor x/y \rfloor y.$$

Consideremos agora o que acontece quando enfileiramos N objetos sem desenfileirar os mesmos. Iremos obter $f = r$, que é a mesma condição de quando a fila está vazia. Por isso, não temos mais a capacidade de diferenciar entre uma fila cheia e uma fila vazia neste caso. Felizmente, este não é um problema muito grande, e existem várias formas de se lidar com ele. Por exemplo, podemos simplesmente insistir que Q nunca pode armazenar mais do que $N - 1$ objetos. Essa regra simples para lidar com uma fila cheia cuida deste problema em nossa implementação, e leva as descrições de pseudocódigo dos métodos principais de uma fila fornecidos no Algoritmo 2.5. Observamos que podemos calcular o tamanho da fila em termos da expressão $(N - f + r) \bmod N$, o que nos fornece o resultado correto tanto na situação "normal" (quando $f \leq r$) como na configuração "que gira" (quando $r < f$).

```
Algoritmo dequeue():
    se isEmpty() então
      lance uma QueueEmptyException
    temp ← Q[f]
    Q[f] ← null
    f ← (f + 1) mod N
    retorne temp

Algoritmo enqueue(o):
    se size() = N – 1 então
      lance uma QueueFullException
    Q[r] ← o
    r ← (r + 1) mod N
```

Algoritimo 2.5 Implementação de uma fila usando um arranjo, o qual é visto de forma circular.

Como em nossa implementação de pilha usando um arranjo, cada um dos métodos na realização do arranjo executa um número fixo de sentenças envolvendo operações aritméticas, comparações e atribuições. Desta forma, cada método nesta implementação executa em tempo $O(1)$.

Da mesma forma, como na implementação da pilha baseada em arranjo, a única desvantagem real na implementação baseada em arranjo é que nós definimos artificialmente a capacidade da fila como sendo um número *N*. Em uma aplicação real, podemos, na verdade, precisar de mais ou menos capacidade do que isso, mas se tivermos uma boa estimativa do número de elementos que estarão na pilha ao mesmo tempo, então a implementação baseada em arranjo será eficiente.

Usando filas para multiprogramação

Multiprogramação é uma maneira de obter uma forma limitada de paralelismo mesmo em computadores que têm apenas uma CPU. Esse mecanismo nos permite ter várias tarefas ou ***threads*** computacionais executando ao mesmo tempo, cada uma sendo responsável por alguma computação específica. A multiprogramação é útil em aplicações gráficas. Por exemplo, uma *thread* pode ser responsável por capturar os cliques do *mouse* enquanto várias outras são responsáveis por mover partes de uma animação na área apropriada da tela. Mesmo se o computador tiver uma única CPU, as diferentes *threads* computacionais podem parecer estar executando simultaneamente porque:

1. a CPU é muito rápida em relação à nossa percepção do tempo;
2. o sistema operacional provê cada *thread* com uma "fatia" do tempo da CPU.

As fatias de tempo atribuídas para cada *thread* diferente se sucedem com tal velocidade que temos a impressão de que as diferentes *threads* estão rodando simultaneamente, em paralelo.

Por exemplo, Java tem um sistema embutido para obter multiprogramação – Java *threads*. Java *threads* são objetos computacionais que podem cooperar e se comunicar entre si para compartilhar outros objetos na memória, a tela do computador ou outros tipos de recursos e dispositivos. A alternância entre as diferentes *threads* em um programa Java ocorre rapidamente, porque cada *thread* tem sua própria pilha Java armazenada na memória da Máquina Virtual Java. A pilha de métodos de cada *thread* contém as variáveis locais e os *frames* para os métodos que cada *thread* está executando no momento. Sendo assim, para trocar de uma *thread T* para outra *thread U*, tudo que a CPU necessita fazer é "lembrar" em que ponto ela deixou a *thread T* antes de alternar para a *thread U*. Já discutimos sobre a forma como isso é feito, a saber, armazenando o valor corrente do contador de instruções de *T*, que é uma referência para a próxima instrução que *T* deve executar no topo da pilha Java de *T*. Armazenando o contador de instruções para cada *thread* ativa no topo de sua pilha Java, a CPU pode retornar ao ponto onde estava em alguma outra *thread U*, restaurando o valor do contador de instruções com o valor que estava armazenado no topo da pilha Java de *U* (e passa a usar a pilha de *U* como a pilha Java "corrente").

Quando projetamos um programa que usa múltiplas *threads*, devemos ter cuidado para não permitir que uma *thread* individual monopolize a CPU. Tal monopolização da CPU pode levar uma aplicação ou *applet* a ***pendurar***, onde ela está tecnicamente executando, mas na realidade não está fazendo nada. Em alguns sistemas operacionais, entretanto, a monopolização da CPU por uma *thread* não é possível. Esses sistemas operacionais utilizam uma fila para alocar o tempo da CPU para as *threads* prontas para executar usando um protocolo ***round-robin***.

A idéia principal do protocolo *round-robin* é armazenar todas as *threads* executáveis em uma fila *Q*. Quando a CPU está pronta para oferecer uma fatia de tempo para

uma *thread*, ela executa uma operação dequeue na fila Q para obter a próxima *thread* disponível para execução; vamos chamá-la de T. Antes da CPU efetivamente iniciar a execução das instruções de T, entretanto, ela inicia um temporizador que executa em *hardware* para se encerrar um certo período de tempo mais tarde. A CPU aguarda então até que (a) a *thread* bloqueie a si mesma (através de um dos métodos de bloqueio citados anteriormente), ou (b) o tempo expire. No último caso, a CPU encerra a execução de T e executa uma operação enqueue para colocar T no final da fila de *threads* prontas para execução. Em qualquer caso, a CPU salva o valor corrente do contador de instruções de T no topo da pilha de métodos de T e processa a próxima *thread* pronta para executar extraindo-a de Q, usando uma operação dequeue. Desta forma, a CPU garante que para cada *thread* pronta para executar é fornecida sua parcela de tempo. Assim, usando uma simples estrutura de dados de fila e um relógio de *hardware*, o sistema operacional pode evitar que a CPU seja monopolizada.

Enquanto essa solução baseda em fila resolve o problema da multiprogramação, podemos mencionar que esta solução é uma simplificação do protocolo usado pela maioria dos sistemas operacionais que usam *round-robin* para a divisão do tempo, pois a maioria dos sistemas atribuem prioridades para as *threads*. Desta forma, eles usam uma ***fila com prioridade*** para implementar as fatias de tempo. Discutiremos filas com prioridade na Seção 2.4.

2.2 Vetores, listas e seqüências

Pilhas e filas armazenam elementos de acordo com uma seqüência linear determinada por operações de atualização que agem no "fim" da seqüência. As estruturas de dados discutidas nesta seção mantêm uma ordenação linear ao mesmo tempo que permitem acessos e atualizações no "meio".

2.2.1 Vetores

Suponha uma dada seqüência linear S que contém n elementos. Podemos nos referir a cada elemento individual e de S usando um inteiro pertencente ao intervalo $[0, n-1]$ que é igual ao número de elementos de S que precede e em S. Definimos a ***colocação*** de um elemento e em S como sendo o número de elementos que antecedem e em S. Sendo assim o primeiro elemento da seqüência está na colocação 0, e o último elemento está na colocação $n-1$.

Observamos que a colocação é similar ao índice de um arranjo, mas isso ***não*** significa que um arranjo deva ser usado para implementar uma seqüência de tal forma que o elemento na colocação 0 seja armazenado no índice 0 do arranjo. A definição de colocação nos oferece uma maneira de se referir ao "índice" de um elemento em uma seqüência sem ter de se preocupar com a forma pela qual ela é implementada. Observamos que a colocação de um elemento pode mudar toda vez que a seqüência é atualizada. Por exemplo, se inserimos um novo elemento no início da seqüência, a colocação dos demais elementos é acrescida em um.

Uma seqüência linear que suporta acesso a seus elementos através de suas colocações é chamada de ***vetor***. Uma colocação é uma notação simples porém poderosa, uma

vez que pode ser usada para especificar onde inserir um novo elemento em um vetor ou onde remover um elemento velho. Por exemplo, podemos determinar a colocação que um elemento novo de um vetor irá ocupar antes de ele ser inserido (por exemplo, insert at rank 2). Podemos também usar colocações para especificar um elemento que deve ser removido (por exemplo, remove the element at rank 2).

O tipo abstrato de dados vetor

Um ***vetor*** S armazenando n elementos suporta os seguintes métodos fundamentais:

elemAtRank(r): retorna o elemento de S com colocação r; uma condição de erro ocorre se $r < 0$ ou $r > n - 1$.

replaceAtRank(r, e): substitui por e o elemento na colocação r e retorna-o; uma condição de erro ocorre se $r < 0$ ou $r > n - 1$.

insertAtRank(r, e): insere um novo elemento e em S na colocação r; uma condição de erro ocorre se $r < 0$ ou $r > n$.

removeAtRank(r): remove de S o elemento na colocação r; uma condição de erro ocorre se $r < 0$ ou $r > n - 1$.

Além disso, um vetor suporta os métodos size() e isEmpty().

Uma implementação simples usando arranjo

Uma escolha óbvia para implementar o TAD vetor é usar um arranjo A, onde $A[i]$ armazena (uma referência para) o elemento na colocação i. Escolhemos o tamanho N do arranjo A, suficientemente grande, e mantemos em uma variável de instância o número $n < N$ de elementos no vetor. Os detalhes de implementação dos métodos do TAD vetor são razoavelmente simples. Para implementar a operação elemAtRank(r), por exemplo, apenas retornamos $A[r]$. Implementações dos métodos insertAtRank(r,e) e removeAtRank(r) são dadas no Algoritmo 2.6. Uma parte importante (e que consome tempo) desta implementação involve o deslocamento de elementos para cima ou para baixo para conservar ocupadas as células contíguas do arranjo. Essas operações de deslocamento são necessárias para manter as regras de sempre armazenar um elemento de colocação i no índice i de A (veja a Figura 2.7 e também o Exercício C-2.5).

```
Algoritmo insertAtRank(r, e):
    para i = n – 1, n – 2, ... , r faça
      A[i + 1] ← A[i]        {arranja espaço para o novo elemento}
    A[r] ← e
    n ← n + 1
Algoritmo removeAtRank(r):
    e ← A[r]  {e é uma variável temporária }
    para i = r, r + 1, ... , n – 2 faça
      A[i] ← A[i + 1]        {preenche o lugar do elemento removido}
    n ← n – 1
    retorne e
```

Algoritmo 2.6 Métodos do TAD vetor em uma implementação usando arranjo.

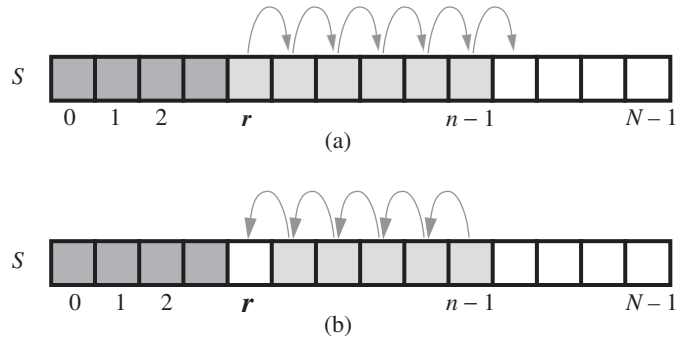


Figura 2.7 Implementação usando arranjo de um vetor S que armazena n elementos: (a) deslocamento para cima visando a abrir espaço para uma inserção na colocação r; (b) deslocamento para baixo visando a uma remoção na colocação r.

A Tabela 2.8 apresenta os tempos de execução dos métodos de um vetor implementado usando um arranjo. É fácil perceber que os métodos isEmpty, size e elemAtRank executam em tempo $O(1)$; porém, os métodos de inserção e remoção podem ser mais demorados. Em especial, insertAtRank(r,e) executa em tempo $\Theta(n)$ no pior caso. Da mesma forma, o pior caso para esta operação ocorre quando $r = 0$, uma vez que todos os n elementos existentes têm de ser deslocados para frente. Um argumento similar se aplica ao método removeAtRank(r), que executa em tempo $O(n)$, porque temos de deslocar para trás $n - 1$ elementos no pior caso ($r = 0$). De fato, supondo que cada colocação possível tem a mesma possibilidade de ser passada por parâmetro para estas operações, a soma dos tempos médios de execução será $\Theta(n)$, pois teremos de deslocar, em média, $n/2$ elementos.

Método	Tempo
size()	$O(1)$
isEmpty()	$O(1)$
elemAtRank(r)	$O(1)$
replaceAtRank(r, e)	$O(1)$
insertAtRank(r, e)	$O(n)$
removeAtRank(r)	$O(n)$

Tabela 2.8 Desempenho no pior caso de um vetor com n elementos implementado usando um arranjo. O gasto de memória é $O(N)$, onde N é o tamanho do arranjo.

Analisando insertAtRank e removeAtRank com mais cuidado, observamos que cada um executa em tempo $O(n - r + 1)$, onde apenas aqueles elementos na colocação r ou maior têm de ser deslocados. Assim, inserir ou remover um item no fim do vetor usando os métodos insertAtRank e removeAtRank, respectivamente, consome tempo $O(1)$ cada. Isto é, a inserção ou remoção de elementos no fim do vetor leva um tempo constante, da mesma forma que inserir ou remover um elemento que dista uma quantidade fixa de células do fim. Ainda, usando a implementação descrita, inserir ou remover um elemento no início do vetor requer o deslocamento de todos os outros elementos de uma posição; logo, leva tempo $\Theta(n)$. Desta forma, existe uma assimetria nesta implementação de vetor – atualizações no final são mais rápidas, enquanto que atualizações no início são mais lentas.

Na verdade, com pequeno esforço, podemos gerar uma implementação usando arranjo para o TAD vetor que obtenha tempo $O(1)$ para inserções e remoções na colocação 0, da mesma forma que no final do vetor. Essa obtenção requer, entretanto, reativar a regra que diz que o elemento de colocação i deve ser armazenado no índice i do arranjo, da mesma forma que podemos usar uma abordagem de arranjo circular como a que usamos na Seção 2.1.2 para implementar uma fila. Deixamos os detalhes dessa implementação como exercício (C-2.5). Além disso, observamos que um vetor também pode ser implementado de forma eficiente usando uma tabela extensível (Seção 1.5.2), o que, de fato, é a implementação *default* do TAD vetor em Java.

2.2.2 Listas

Usar colocações não é a única forma de nos referirmos ao local onde um elemento aparece em uma lista. Podemos, alternativamente, implementar uma lista S de forma que cada elemento seja armazenado em um objeto ***nodo*** especial com referências para os nodos que o antecedem e o sucedem na lista. Neste caso, pode ser mais natural e eficiente usar um nodo em vez de uma colocação para identificar onde acessar e atualizar uma lista. Nesta seção, iremos explorar uma forma de abstrair o conceito de "lugar" como nodo de uma lista.

Posições e operações usando nodos

Gostariamos de definir métodos para uma lista S que recebessem nodos da lista como parâmetro e os usassem como tipo de retorno. Por exemplo, podemos definir um método hipotético removeAtNode(v) que remove o elemento de S armazenado no nodo v da lista. O uso de um nodo como parâmetro nos permite remover um nodo em tempo $O(1)$, simplesmente indo direto ao local onde o nodo está armazenado e, então, "desconectando-o" através de atualizações nas referências de seus vizinhos.

Definir métodos de um TAD lista adicionando tais operações baseadas em nodos nos leva à questão sobre quanta informação estamos expondo sobre a implementação de nossa lista. Certamente, nos é desejável poder usar essa implementação sem revelar esses detalhes para o usuário. Igualmente, não queremos que o usuário seja capaz de modificar a estrutura interna da lista sem conhecimento. Para abstrair e unificar as diferentes formas de armazenar elementos nas várias implementações de uma lista, introduzimos o conceito de ***posição***, que formaliza a noção intuitiva de "lugar" de um elemento relativo a outros em uma lista.

De maneira a expandir com segurança o conjunto de operações para listas, abstraímos a noção de "posição", que nos permite aproveitar a eficiência das implementações de lista que usam nodos sem violar os princípios de projeto orientado a objetos. Neste *framework*, percebemos uma lista como um contêiner de elementos que armazenam cada elemento em uma posição e que mantém essas posições organizadas de maneira linear. Uma posição é também um tipo abstrato de dados que suporta um método simples:

element(): retorna o elemento armazenado nesta posição.

Uma posição é sempre definida ***relativamente***, ou seja, em relação a seus vizinhos. Em uma lista, uma posição p estará sempre "depois" de uma certa posição q e "antes" de uma posição s (a menos que p seja a primeira ou a última posição). Uma posição p, associada com um elemento e em uma lista S, não muda, mesmo que a colocação de e se altere em S, a menos que removamos e explicitamente (e, então, destruímos a posição p). Além disso, a posição p não muda mesmo que o elemento e armaze-

nado em p seja substituído ou trocado por outro. Esses fatos sobre posições nos permitem definir um conjunto rico em métodos de lista usando posições que recebem a posição dos objetos como parâmetro e provêem objetos de posição como valores de retorno.

O tipo abstrato de dados lista

Usando o conceito de posição para encapsular a idéia de "nodo" em uma lista, podemos definir outro tipo de TAD seqüência chamado simplesmente de ***lista***. Esse TAD suporta os seguintes métodos para uma lista S:

first(): retorna a posição do primeiro elemento de S; um erro ocorre se S estiver vazio.

last(): retorna a posição do último elemento de S; um erro ocorre se S estiver vazio.

isFirst(p): retorna um valor booleano que indica quando a posição fornecida é a primeira da lista.

isLast(p): retorna um valor booleano que indica quando a posição fornecida é a última da lista.

before(p): retorna a posição do elemento de S que precede aquele que se encontra na posição p; um erro ocorre se p for a primeira posição.

after(p): retorna a posição do elemento de S que segue aquele que se encontra na posição p; um erro ocorre se p for a última posição.

Os métodos apresentados permitem que nos referenciemos a posições relativas da lista, iniciando pelo começo ou pelo fim, e tendo a habilidade de se mover incrementalmente para cima ou para baixo na lista. Essas posições podem ser entendidas intuitivamente como nodos de uma lista, mas note que não existem referências específica para objetos nodo nem ligações com os nodos anterior ou posterior nestes métodos. Além desses métodos e dos métodos genéricos size e isEmpty, também incluímos os seguintes métodos de atualização para o TAD lista:

replaceElement(p,e): substitui o elemento na posição p por e, retornando o elemento que ocupava originalmente a posição p.

swapElements(p,q): troca os elementos armazenados nas posições p e q, de maneira que o elemento que está na posição p é movido para a posição q, e o elemento que está na posição q é movido para a posição p.

insertFirst(e): insere um novo elemento e em S, que passa a ser o primeiro elemento.

insertLast(e): insere um novo elemento e em S, que passa a ser o último elemento.

insertBefore(p,e): insere um novo elemento e em S antes da posição p de S.

insertAfter(p,e): insere um novo elemento e em S depois da posição p de S; um erro ocorre se p estiver na última posição.

remove(p): remove de S o elemento da posição p.

O TAD lista nos permite visualizar uma coleção ordenada de objetos em função de seus lugares, sem se preocupar com a forma pela qual esses lugares são representados. Observamos também que existe alguma redundância no repertório de operações deste TAD. Na verdade, podemos executar as operações isFirst(p) verificando quando p é igual à posição retornada por first(). Podemos executar, também, a operação insertFirst(e) executando a operação insertBefore(first(),e). A redundância dos métodos pode ser vista como um conjunto de atalhos.

Uma condição de erro ocorre se uma posição passada por parâmetro para uma das operações da lista não for válida. Razões para uma posição p ser inválida incluem a mesma ser nula, ser uma posição de uma lista diferente ou ser uma posição previamente removida da lista.

O TAD lista, com sua noção embutida de posição, é útil em uma grande quantidade de situações. Por exemplo, um simples editor de textos inclui a noção de inserção e remoção por posição, uma vez que tais editores tipicamente executam todas as atualizações relativas a um ***cursor***, que representa a posição atual na lista de caracteres do texto sendo editado.

Implementação de lista encadeada

A estrutura de dados ***lista encadeada*** nos possibilita uma grande quantidade de operações, incluindo inserção e remoção em vários lugares, executando em tempo $O(1)$. Um nodo em uma ***lista simplesmente encadeada*** armazena em uma conexão ***next*** uma referência para o próximo nodo da lista. Desta forma, uma lista simplesmente encadeada só pode ser percorrida em uma única direção – da cabeça para a cauda. Um nodo em uma ***lista duplamente encadeada***, por outro lado, armazena duas referências – uma conexão ***next***, que aponta para o próximo nodo da lista, e uma conexão ***prev***, que aponta para o nodo anterior da lista. Desta forma, uma lista duplamente encadeada pode ser percorrida em ambas as direções. Ser capaz de determinar os nodos anterior e posterior de qualquer nodo da lista simplifica enormemente sua implementação; então vamos supor que estamos usando uma lista duplamente encadeada para implementar o TAD lista.

Para simplificar atualizações e pesquisas, é conveniente acrescentar nodos especiais nas extremidades da lista: um nodo ***header*** antes do início da lista e um nodo ***trailer*** após o último da lista. Esses nodos "simulados" ou ***sinalizadores*** não armazenam elementos. O *header* tem uma referência *next* válida mas uma referência *prev* nula, enquanto que o *trailer* tem uma referência *prev* válida mas uma referência *next* nula. Uma lista duplamente encadeada com essas sentinelas pode ser vista na Figura 2.9. Observamos que um objeto lista encadeada necessita simplesmente armazenar estas duas sentinelas e um contador de tamanho que mantém o número de elementos (sem contar as sentinelas) na lista.

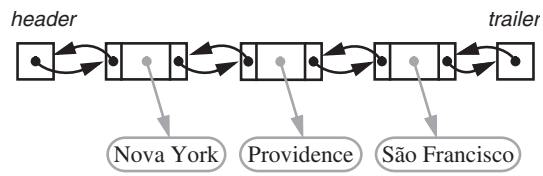


Figura 2.9 Uma lista duplamente encadeada com sentinelas, *header* e *trailer*, marcando os extremos da lista. Uma lista vazia terá essas sentinelas apontando uma para a outra.

Podemos fazer simplesmente os nodos de uma lista encadeada implementar o TAD posição definindo um método element(), que retorne o elemento armazenado no nodo. Entretanto, os próprios nodos atuam como posições.

Considere como podemos implementar o método insertAfter(p,e), que insere um elemento e depois da posição p. Criamos um novo nodo v para guardar o elemento e, ligamos v ao seu lugar na lista e então atualizamos as referências *next* e *prev* dos novos vizinhos de v. Esse método é apresentado em pseudocódigo no Algoritmo 2.10 e ilustrado na Figura 2.11.

Algoritmo insertAfter(p,e):
  Cria um novo nodo v
  v.element ← e
  v.prev ← p {liga v com seu predecessor}
  v. next ← p.next {liga v com seu sucessor}
  (p.next).prev ← v {liga o antigo sucessor de p com v}
  p.next ← v {liga p com seu novo sucessor v}
  retorne v {a posição do elemento e}

Algoritmo 2.10 Inserção de um elemento e depois da posição p em uma lista encadeada.

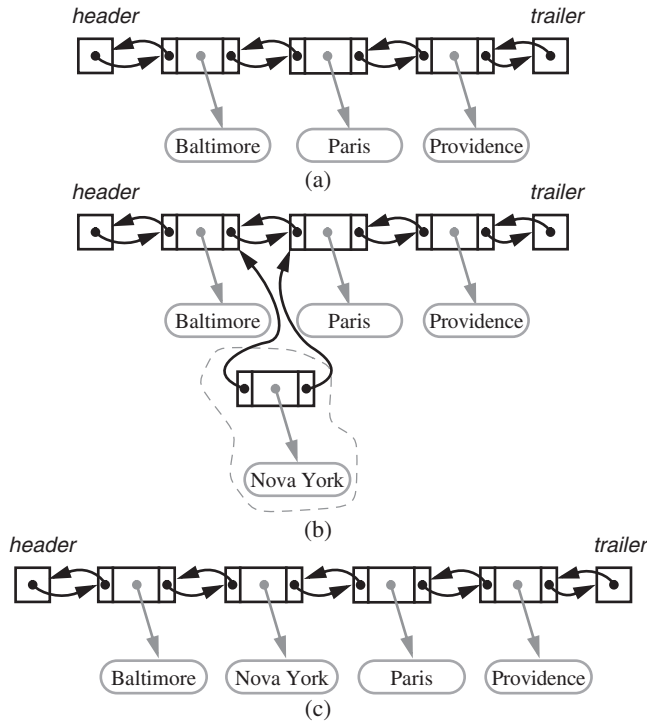


Figura 2.11 Acrescentando um novo nodo após a posição de "Baltimore": (a) antes da inserção; (b) criando um novo nodo v e ligando-o; (c) após a inserção.

Os algoritmos para os métodos insertBefore, insertFirst e insertLast são similares ao do método insertAfter; deixamos seus detalhes como exercício (R-2.1). Em seguida considere o método remove(p), que remove o elemento e armazenado na posição p.

Para executar essa operação, ligamos os dois vizinhos de p de maneira que referenciem um ao outro como novos vizinhos – desconectando p. Observamos que, depois de p ser desconectado, nenhum nodo estará mais apontando para p; desta forma, a sistema de coleta de lixo pode recuperar o espaço de p. Esse algoritmo é apresentado no Algoritmo 2.12 e ilustrado na Figura 2.13. Lembrando nosso uso de sentinelas, observe que este algoritmo funciona mesmo se p for o primeiro, o último ou apenas uma posição qualquer da lista.

Algoritmo remove(p):
  $t \leftarrow p$.element {uma variável temporária para armazenar o valor a ser retornado}
  (p.prev).next $\leftarrow p$.next {desconecta p}
  (p.next).prev $\leftarrow p$.prev
  p.prev $\leftarrow$ **null**
  p.next $\leftarrow$ **null**
  retorne t

Algoritmo 2.12 Remoção do elemento e armazenado na posição p de uma lista encadeada.

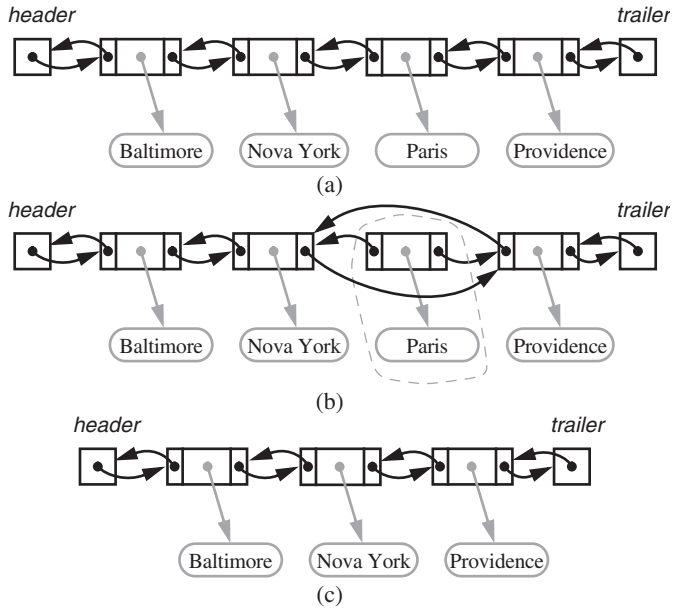


Figura 2.13 Remoção do objeto armazenado na posição de "Paris": (a) antes da remoção; (b) desconectando o nodo antigo; (c) após a remoção (e coleta de lixo).

2.2.3 Seqüências

Nesta seção, definiremos o TAD seqüência genérica, que inclui todos os métodos dos TADs lista e vetor. Este TAD, entretanto, permite o acesso aos seus elementos usando tanto colocações como posições, e é a mais versátil das estruturas de dados para uma grande gama de aplicações.

O tipo abstrato de dados seqüência

Uma ***seqüência*** é um TAD que suporta todos os métodos dos TADs vetor (apresentado na Seção 2.2.1) e lista (discutido na Seção 2.2.2), mais os dois métodos "de conexão", a seguir, que permitem relacionar colocações e posições.

atRank(r): retorna a posição do elemento na colocação r.

rankOf(p): retorna a colocação do elemento na posição p.

Uma seqüência genérica pode ser implementada usando-se tanto uma lista duplamente encadeada como um arranjo, com as restrições naturais a esses dois tipos de implementação. A Tabela 2.14 compara os tempos de execução das implementações de um TAD seqüência usando um arranjo (organizado de forma circular) e usando uma lista duplamente encadeada.

Operações	Arranjos	Lista
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last, before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore,	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Tabela 2.14 Comparação dos tempos de execução dos métodos de uma seqüência implementada usando-se um arranjo (usado de forma circular) ou uma lista duplamente encadeada. Denotamos por n o número de elementos da seqüência no momento em que a operação é executada. O espaço gasto é $O(n)$ para a implementação usando lista encadeada e $O(N)$ para a implementação baseada em arranjo.

Resumindo essa tabela, percebemos que a implementação usando arranjo é superior à baseada em lista encadeada quando se trata de operações baseadas em colocações (atRank, rankOf e elemAtRank), e tem perfomance equivalente nas demais operações de acesso. Considerando as operações de atualização, a implementação usando lista encadeada supera a implementação baseada em arranjo nas operações que utilizam posições (insertAfter, insertBefore e remove). Tanto a implementação que usa arranjo como a que usa lista encadeada tem o mesmo pior caso em relação à performance no que se refere às operações de atualização usando colocações (insertAtRank e removeAtRank), mas por diferentes razões. No que se refere às operações de atualização (insertFirst e insertLast), as duas implementações têm performances semelhantes.

Considerando o gasto de memória, observamos que o arranjo requer espaço $O(N)$, onde N é o tamanho do arranjo (a menos que utilizemos um arranjo de tamanho variável), enquanto que uma lista duplamente encadeada ocupa um espaço $O(n)$, onde n é o número de elementos na seqüência. Como n é menor ou igual a N, isso implica que o

espaço assintótico gasto por uma lista encadeada é superior ao de um arranjo de tamanho fixo, embora exista um pequeno fator de sobrecarga constante que é maior para listas encadeadas, uma vez que arranjos não necessitam de conexões para manter a ordem de suas células.

As implementações baseadas em arranjo e lista encadeada têm cada uma suas vantagens e desvantagens. A mais correta para uma aplicação particular depende dos tipos de operações que serão usadas no espaço de memória disponível. Projetar o TAD seqüência de maneira a torná-lo independente da forma como é implementado nos permite trocar facilmente de implementação, possibilitando escolher a melhor para cada tipo de aplicação com poucas alterações no programa.

Iteradores

Uma computação típica em um vetor, lista ou seqüência é caminhar através de seus elementos em ordem, um de cada vez, por exemplo, procurando por um elemento específico.

Um ***iterador*** é um padrão de projeto de *software* que abstrai o processo de percorrer uma coleção de elementos um de cada vez.

Um iterador consiste em uma seqüência S, uma posição corrente em S e uma forma de avançar para a próxima posição de S e torná-la a posição atual. Desta forma, o iterador estende o conceito do TAD posição introduzido na Seção 2.2.2. De fato, uma posição pode ser entendida como um iterador que não vai a ligar nenhum. Um iterador encapsula os conceitos de "lugar" e "próximo" em uma coleção de objetos.

Definimos o TAD iterador como suportando os dois métodos que seguem:

hasNext: testa se existem elementos no iterador.

nextObject: retorna e remove o próximo elemento do iterador.

Observamos que este TAD tem a noção de elemento "atual" em um caminhamento através de uma seqüência. O primeiro elemento do iterador é retornado na primeira chamada ao método nextObject, assumindo, é claro, que o iterador contenha pelo menos um elemento.

Um iterador provê um esquema unificado para acessar todos os elementos de um contêiner (uma coleção de objetos) de uma forma independente da organização específica da coleção. Um iterador para uma seqüência deve retornar os elementos de acordo com sua ordem linear.

2.3 Árvores

Uma ***árvore*** é um tipo abstrato de dados que armazena elementos de forma hierárquica. Com a exceção do elemento do topo, cada elemento de uma árvore tem um elemento ***pai*** e zero ou mais elementos ***filhos***. Uma árvore normalmente é visualizada desenhando-se os elementos dentro de círculos ou de retângulos, e desenhando as conexões entre pais e filhos usando linhas retas (veja a Figura 2.15). Tipicamente chamamos o elemento topo de ***raiz*** da árvore, que é desenhado, porém, como o elemento mais alto, com os demais abaixo dele (ao contrário de uma árvore da botânica).

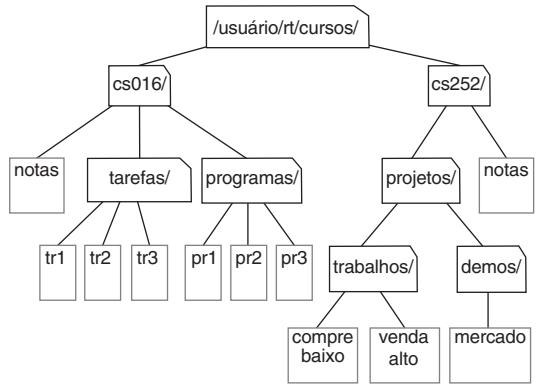


Figura 2.15 Uma árvore representando parte de um sistema de arquivos.

Uma ***árvore*** *T* é um conjunto de ***nodos*** que armazenam elementos em um relacionamento ***pai-filho*** com as seguintes propriedades:

- *T* tem um nodo especial *r*, chamado a ***raiz*** da árvore *T*;
- cada nodo *v* de *T* diferente de *r* tem um nodo ***pai*** *u*.

Observamos que, de acordo com a definição anterior, uma árvore não pode ser vazia, uma vez que ela deve ter pelo menos um nodo, a raiz. É possível alterar essa definição para permitir árvores vazias, mas adotaremos a convenção de que uma árvore sempre tem uma raiz de maneira a simplificar nossa apresentação e evitar ter de estar sempre tratando o caso especial de árvore vazia em nossos algoritmos.

Se o nodo *u* é pai do nodo *v*, então dizemos que *v* é ***filho*** de *u*. Dois nodos que são filhos do mesmo pai são ditos ***irmãos***. Um nodo é ***externo*** se não tiver filhos, e é ***interno*** se tiver um ou mais filhos. Nodos externos são também conhecidos como ***folhas***. A ***subárvore*** de *T* ***enraizada*** no nodo *v* é a árvore que consiste de todos os descendentes de *v* em *T* (incluindo o próprio *v*). O ***ancestral*** de um nodo pode ser tanto seu ancestral direto como um ancestral do pai do nodo. Da mesma forma, dizemos que um nodo *v* é ***descendente*** de um nodo *u* se *u* for um ancestral de *v*.

Exemplo 2.2: *na maioria dos sistemas operacionais, arquivos são organizados de forma hierárquica em diretórios aninhados (também chamados de pastas), que são apresentados para o usuário sob a forma de uma árvore (veja a Figura 2.15). Mais especificamente, os nodos internos da árvore são associados com diretórios, e os nodos externos são associados com arquivos comuns. No sistema operacional UNIX/Linux, a raiz da árvore é apropriadamente chamada de "diretório raiz", e é representada pelo símbolo "/". Corresponde ao ancestral de todos os diretórios e arquivos de um sistema de arquivos UNIX/Linux.*

Uma árvore está ***ordenada*** se existe uma ordem linear definida para cada filho de cada nodo; isto é, podemos identificar o filho de um nodo como sendo o primeiro, segundo, terceiro e assim por diante. Árvores ordenadas normalmente indicam uma relação linear entre os irmãos listando-os, na ordem correta, em uma seqüência ou iterador.

Exemplo 2.3: *um documento estruturado, tal como um livro, é organizado hierarquicamente como uma árvore cujos nodos internos são capítulos, seções e subseções, e cujos nodos externos são parágrafos, tabelas, figuras, bibliografia e assim por diante (veja a Figura 2.16). Podemos, na verdade, considerar ainda a expansão da árvore para representar parágrafos compostos por sentenças, sentenças compostas por palavras e palavras por caracteres. Em qualquer caso, porém, esta árvore é um exemplo de árvore ordenada, porque existe um ordem bem definida entre os filhos de cada nodo.*

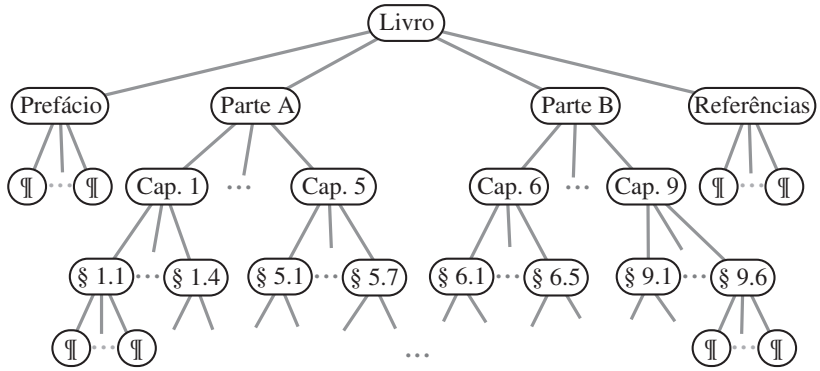


Figura 2.16 Uma árvore associada a um livro.

Uma ***árvore binária*** é uma árvore ordenada na qual cada nodo tem no máximo dois filhos. Uma árvore binária é dita ***própria*** se todo nodo interno tiver dois filhos. Cada um dos nodos internos de uma árvore binária é rotulado como sendo o ***filho da esquerda*** ou o ***filho da direita***. Estes filhos são ordenados de maneira que o filho da esquerda venha antes do filho da direita. A subárvore enraizada no filho da esquerda ou no filho da direita de um nodo interno v é chamada de ***subárvore*** ou ***subárvore direita*** de v, respectivamente. Convencionamos neste livro que, a menos que se coloque algo em contrário, toda árvore binária é uma árvore binária própria. Naturalmente, mesmo uma árvore binária própria é também uma árvore genérica, com a propriedade de que cada nodo interno tenha no máximo dois filhos. Árvores binárias tem diversas aplicações interessantes, incluindo as que seguem.

Exemplo 2.4: *uma expressão pode ser representada por uma árvore cujos nodos externos são associados com variáveis ou constantes, e cujos nodos internos são associados com um dos operadores* $+, -, \times$ *e* $/$ *(veja a Figura 2.17). Cada nodo em tal árvore tem um valor associado ao mesmo.*

- *se um nodo for externo, então seu valor é o de uma variável ou constante;*
- *se um nodo for interno, então seu valor é definido pela aplicação de sua operação aos valores de seus filhos.*

Árvores que representam expressões aritméticas são árvores binárias próprias, uma vez que cada operador $+, -, \times$ *e* $/$ *tem exatamente dois operandos. Naturalmente, se admitirmos operadores unários, tais como negação* $(-)$, *como em "–x", então teremos uma árvore binária imprópria.*

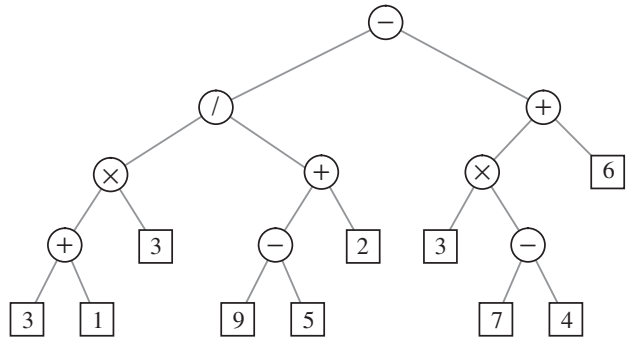


Figura 2.17 Uma árvore binária representando uma expressão aritmética. Esta árvore representa a expressão ((((3+1) × 3) / ((9 – 5) + 2)) – ((3 × (7 – 4)) + 6)). O valor associado ao nodo interno rotulado com "/" é 2.

2.3.1 O tipo abstrato de dados árvore

O TAD árvore armazena elementos em posições, que, como as posições de uma lista, são definidas em relação às posições vizinhas. As ***posições*** em uma árvore são seus ***nodos***, e as posições vizinhas satisfazem as relações pai-filho que definem uma árvore válida. Desta forma, usaremos as expressões "posição" e "nodo" com o mesmo significado no caso de árvores. Da mesma forma que as posições de uma lista, um objeto posição de uma árvore suporta o método element(), que retorna o objeto armazenado nesta posição. O potencial real dos nodos posição de uma árvore, entretanto, está nos seguintes ***métodos de acesso*** do TAD árvore:

root(): retorna a raiz da árvore.

parent(v): retorna o pai do nodo v; um erro ocorre se v for a raiz.

children(v): retorna um iterador sobre os filhos do nodo v.

Se uma árvore T é ordenada, então o iterador children(v) fornece acesso aos filhos de v na ordem. Se v é um nodo externo, então children(v) é um iterador vazio.

Além disso, também incluímos os seguintes ***métodos de consulta***:

isInternal(v): testa se um nodo v é interno.

isExternal(v): testa se um nodo v é externo.

isRoot(v): testa se um nodo v é a raiz.

Existem também alguns métodos que uma árvore pode suportar que não estão necessariamente relacionados com a estrutura da árvore. Tais ***métodos genéricos*** incluem os seguintes:

size(): retorna o número de nodos em uma árvore.

elements(): retorna um iterador sobre todos os elementos armazenados nos nodos de uma árvore.

positions(): retorna um iterador sobre todos os nodos da árvore.

swapElements(v,w): troca os elementos armazenados nos nodos v e w.

replaceElement(v,e): substitui por e e retorna o elemento armazenado no nodo v.

Não definimos nenhum método específico para atualização de uma árvore aqui. Em vez disso, reservaremos o potencial para definir os diferentes métodos de atualização juntamente com aplicações específicas de árvores.

2.3.2 Caminhamento em árvores

Nesta seção, apresentamos algoritmos que executam computações em uma árvore acessando-a através dos métodos do TAD árvore.

Pressupostos

De maneira a analisar os tempos de execução dos algoritmos que usam árvores, partiremos dos seguintes pressupostos sobre os tempos de execução dos métodos do TAD árvore:

- Os métodos de acesso root() e parent() consomem tempo $O(1)$.
- Os métodos de consulta isInternal(v), isExternal(v) e isRoot(v) consomem tempo $O(1)$, da mesma forma.
- O método de acesso children(v) consome tempo $O(c_v)$, onde c_v é o número de filhos de v.
- Os métodos genéricos swapElements(v,w) e replaceElement(v,e) consomem tempo $O(1)$.
- Os métodos genéricos elements() e positions(), que retornam iteradores, consomem tempo $O(n)$, onde n é o número de nodos da árvore.
- Sobre os iteradores retornados pelos métodos elements(), positions() e children(v), os métodos hasNext(), nextObject() ou nextPosition() consomem tempo $O(1)$ cada um.

Na Seção 2.3.4, apresentaremos estruturas de dados para árvores que satisfazem os pressupostos listados. Antes de descrever como implementar o TAD lista usando uma estrutura de dados concreta, entretanto, vamos descrever como podemos usar os métodos de um TAD árvore para resolver alguns problemas interessantes de árvores.

Profundidade e altura

Seja v um nodo da árvore T. A ***profundidade*** de v é o número de ancestrais de v excluindo o próprio v. Observamos que esta definição implica que a profundidade da raiz de T seja 0. A profundidade de um nodo v também pode ser calculada recursivamente como segue:

- se v for a raiz, então a profundidade de v é 0;
- em qualquer outro caso, a profundidade de v é 1 mais a profundidade do pai de v.

Baseado na definição anterior, o algoritmo recursivo depth, apresentado no Algoritmo 2.18, calcula a profundidade de um nodo v de T chamando a si mesmo, recursivamente, sobre o pai de v e acrescentando 1 ao valor retornado.

```
Algoritmo depth(T,v):
    se T.isRoot(v) então
      retorne 0
    senão
      retorne 1 + depth(T,T.parent(v))
```

Algoritmo 2.18 Algoritmo depth para calcular a profundidade de um nodo v em uma árvore T.

O tempo de execução do algoritmo depth(T,v) é $O(1 + d_v)$, onde d_v denota a profundidade do nodo v na árvore T, porque o algoritmo executa um passo recursivo que consome um tempo constante para cada ancestral de v. Logo, no pior caso, o algortimo depth executa em tempo $O(n)$, onde n é o total de nodos na árvore T, uma vez que alguns nodos podem estar próximos a esta profundidade na árvore. Embora este tempo de execução seja uma função do tamanho da entrada, é mais exato caracterizar o tempo de execução em termos do parâmetro d_v, uma vez que este será quase sempre muito menor do que n.

A ***altura*** da árvore T é igual à profundidade máxima de um nodo externo de T. Apesar desta definição ser correta, ela não nos leva a um algoritmo eficiente. Na verdade, se formos aplicar o algoritmo de determinação de profundidade a cada um dos nodos de T, iremos derivar um algoritmo que consome tempo $O(n^2)$ para calcular a altura de T. Podemos fazer melhor, entretanto, usando a seguinte definição recursiva para a altura de um nodo v em uma árvore T:

- se v for um nodo externo, então a altura de v é 0;
- em qualquer outro caso, a altura de v é 1 mais a maior altura dos filhos de v.

A ***altura*** da árvore T é a altura da raiz de T.

O algoritmo height, apresentado no Algoritmo 2.19, calcula a altura da árvore T de uma forma eficiente usando a definição recursiva de altura apresentada. O algoritmo é expresso por um método recursivo height(T,v) que calcula a altura da subárvore de T enraizada em um nodo v. A altura da árvore T é obtida chamando-se height(T,T.root()).

```
Algoritmo height(T,v):
    se T.isExternal(v) então
      retorne 0
    senão
      h = 0
      para cada w ∈ T.children(v) faça
        h = max(h,height(T,w))
      retorne 1 + h
```

Algoritmo 2.19 Algoritmo height para calcular a altura de uma subárvore da árvore T enraizada no nodo v.

O algoritmo height é recursivo e, se for inicialmente chamado na raiz de T, ele será por fim acionado uma vez em cada nodo de T. Sendo assim, podemos determinar o tempo de execução deste método usando um argumento de amortização em que primeiramente determinamos o tempo gasto em cada nodo (na parte não-recursiva), e então somamos esse tempo limite a todos os nodos. O cálculo do iterador children(v) consome tempo $O(c_v)$, onde c_v denota a quantidade de filhos do nodo v. Da mesma forma, o laço **para** tem c_v iterações, e cada iteração do laço consome tempo $O(1)$ mais o tempo da chamada recursiva sobre um filho de v. Logo, o algoritmo heigth gasta tem-

po $O(1 + c_v)$ em cada nodo v, e seu tempo de execução é $O(\sum_{v \in T}(1 + c_v))$. De maneira a completar a análise, faremos uso da propriedade que segue.

Teorema 2.5: *seja T uma árvore com n nodos, e faça c_v denotar o número de filhos de um nodo v de T. Então*

$$\sum_{v \in T} c_v = n - 1.$$

Prova: Cada nodo de T, com exceção da raiz, é um filho de outro nodo; logo, contribui com uma unidade para o somatório $\sum_{v \in T} c_v$. ■

Pelo Teorema 2.5, o tempo de execução do Algoritmo height quando chamado a partir da raiz de T é $O(n)$, onde n é o número de nodos de T.

Um ***caminhamento*** em uma árvore T é uma forma sistemática de acessar ou "visitar" todos os nodos de T. Apresentaremos em seguida os esquemas básicos de caminhamento para árvores, chamados de prefixado e pós-fixado.

Caminhamento prefixado

No caminhamento ***prefixado*** de uma árvore T, a raiz de T é visitada em primeiro lugar, e então as subárvores enraizadas nos seus filhos são percorridas recursivamente. Se a árvore é ordenada, então as subárvores são percorridas de acordo com a ordem dos filhos. A ação específica associada com a "visita" de um nodo v depende da aplicação do caminhamento, e pode envolver qualquer coisa desde incrementar um contador até algum cálculo complexo sobre v. O pseudocódigo para o caminhamento prefixado em uma subárvore enraizada no nodo v é apresentado no Algoritmo 2.20. Inicialmente chamamos essa rotina como preorder(T,T.root()).

Algoritmo preorder(T,v):
  execute a ação de "visita" para o nodo v
  para cada filho w de v **faça**
    percorra recursivamente a subárvore enraizada em w chamando preorder(T,w)

Algoritmo 2.20 Algoritmo preorder.

O algoritmo de caminhamento prefixado é útil para produzir uma ordenação linear dos nodos de uma árvore quando é necessário que os pais apareçam antes dos filhos na ordenação. Tais ordenações têm diversas aplicações diferentes; exploramos uma instância simples de tais aplicações no exemplo a seguir.

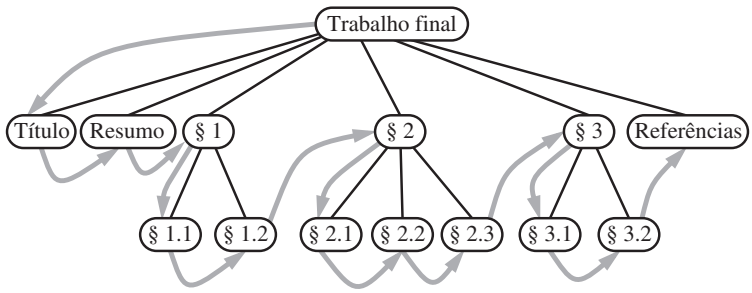


Figura 2.21 Caminhamento prefixado em uma árvore ordenada.

Exemplo 2.6: *o caminhamento prefixado sobre a árvore associada a um documento, como no Exemplo 2.3, examina todo o documento seqüencialmente, do início até o fim. Se os nodos externos forem removidos antes do caminhamento, então o caminhamento examina o índice do documento (veja a Figura 2.21).*

A análise do caminhamento prefixado é, na verdade, parecida com a do algoritmo height fornecido anteriormente. Em cada nodo v, a parte não-recursiva do algoritmo de caminhamento prefixado requer tempo $O(1 + c_v)$, onde c_v é o número de filhos de v. Logo, pelo Teorema 2.5, o tempo total de execução do caminhamento prefixado de T é $O(n)$.

Caminhamento pós-fixado

Outro caminhamento importante sobre árvores é o caminhamento ***pós-fixado***. Este algoritmo pode ser visto como o oposto do algoritmo prefixado, porque ele percorre recursivamente as árvores enraizadas nos filhos da raiz primeiramente, e então visita a raiz. Entretanto, é parecido com o caminhamento prefixado na medida em que o usamos para resolver um problema específico especializando a ação associada com a "visita" a um nodo v. Ainda, da mesma forma que o caminhamento prefixado, se a árvore está ordenada, fazemos as chamadas recursivas sobre os filhos de um nodo v de acordo com sua ordem específica. O pseudocódigo para o caminhamento pós-fixado é fornecido no Algoritmo 2.22.

Algoritmo postorder(T,v):
  para cada filho w de v **faça**
    recursivamente percorra a subárvore enraizada em w chamando postorder(T,v)
  execute a ação de "visita" para o nodo v

Algoritmo 2.22 Método postorder.

O nome do caminhamento pós-fixado vem do fato que este método de caminhamento vai visitar o nodo v depois de ter visitado todos os nodos da subárvore enraizada em v (veja a Figura 2.23).

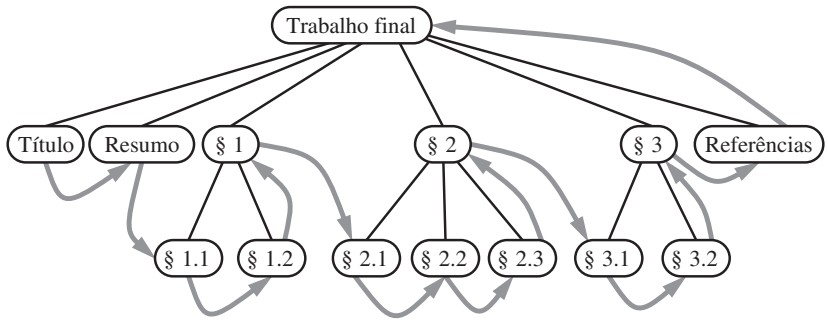


Figura 2.23 Caminhamento pós-fixado da árvore ordenada da Figura 2.21.

A análise do tempo de execução de um caminhamento pós-fixado é análogo ao de um caminhamento prefixado. O tempo total gasto nas porções não-recursivas do algoritmo é proporcional ao tempo gasto visitando os filhos de cada nodo da árvore. Logo, um caminhamento pós-fixado de uma árvore T com n nodos leva tempo $O(n)$, supon-

do que visitar cada nodo leva tempo $O(1)$. Isto é, o caminhamento pós-fixado executa em um tempo linear.

O método de caminhamento pós-fixado é útil para resolver problemas em que queremos calcular alguma propriedade para cada nodo de v em uma árvore, mas o cálculo da propriedade para v requer que já tenhamos calculado a mesma propriedade para os filhos de v. Tais aplicações são demonstradas no próximo exemplo.

Exemplo 2.7: *considere a árvore T de um sistema de arquivos onde os nodos externos representam arquivos e os nodos internos representam diretórios (Exemplo 2.2). Suponha que desejamos calcular o espaço em disco usado por um diretório, que é recursivamente fornecido pela soma de:*

- *tamanho do diretório propriamente dito;*
- *tamanhos dos arquivos nos diretórios;*
- *espaço usado pelos diretórios filhos.*

(Veja Figura 2.2.4.) Este cálculo pode ser feito usando-se um caminhamento pós-fixado sobre a árvore T. Após as subárvores de um nodo interno v terem sido percorridas, calculamos o espaço usado por v somando o tamanho do diretório v, propriamente dito, com os dos arquivos contidos em v mais o espaço usado por cada nodo interno filho de v, que foram calculados pelo caminhamento pós-fixado recursivo sobre os filhos de v.

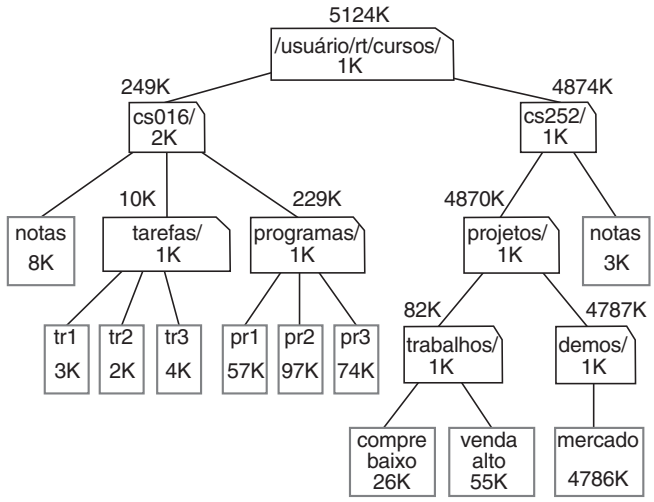


Figura 2.24 Árvore da Figura 2.15 representando um sistema de arquivos, apresentando o nome e o tamanho dos arquivos/diretórios associados dentro de cada nodo e o espaço em disco usado pelo diretório associado sobre cada nodo interno.

Apesar de os caminhamentos prefixado e pós-fixado serem as formas mais comuns de se visitar os nodos de uma árvore, é possível imaginar outros caminhamentos. Por exemplo, podemos percorrer uma árvore de maneira a visitar todos os nodos a uma profundidade d antes de visitar os nodos na profundidade $d + 1$. Tal caminhamento pode ser implementado, por exemplo, usando uma fila, enquanto que os caminhamentos prefixado e pós-fixado usam uma pilha (esta pilha está implicita no uso da recursão para descrever esses métodos, mas podemos tornar este uso explícito para evitar a recur-

são). Além disso, as árvores binárias, que discutiremos em seguida, suportam um método de caminhamento adicional, conhecido como caminhamento interfixado.

2.3.3 Árvores binárias

Um tipo de árvore particularmente interessante é a árvore binária. Como mencionamos na Seção 2.3, uma ***árvore binária*** própria é uma árvore ordenada na qual cada nodo interno tem exatamente dois filhos. Estabelecemos como convenção que, a menos que se diga algo em contrário, presume-se que as árvores binárias são sempre próprias. Note-se que nossa convenção para árvores binárias não implica perda de generalidade, na medida em que pode-se converter facilmente qualquer árvore binária imprópria em uma árvore binária própria, como exploramos no Exercício C-2.14. Mesmo sem essa conversão, podemos considerar uma árvore binária imprópria como própria, simplesmente entendendo os nodos externos faltantes como "nodos nulos" ou lugares que contam como nodos.

O tipo abstrato de dados árvore binária

Como tipo abstrato de dados, uma árvore binária é uma especialização de uma árvore que suporta três métodos de acesso adicionais:

leftChild(v): retorna o filho da esquerda de v; uma condição de erro ocorre se v for um nodo externo.

rightChild(v): retorna o filho da direita de v; uma condição de erro ocorre se v for um nodo externo.

sibling(v): retona o irmão do nodo v; uma condição de erro ocorre se v for a raiz.

Observamos que esses métodos devem ter condições de erro adicionais se estivermos lidando com árvores binárias impróprias. Por exemplo, em uma árvore binária imprópria, um nodo interno pode não ter o filho da esquerda ou o filho da direita. Não incluímos aqui nenhum método para atualizar uma árvore binária, pois tais métodos podem ser criados na medida do necessário no contexto de necessidades específicas.

Propriedades de árvores binárias

Denotamos o conjunto de todos os nodos de uma árvore T que se encontram no mesmo nível d como sendo o ***nível*** de T. Em uma árvore binária, o nível 0 tem um nodo (a raiz), o nível 1, no máximo dois nodos (os filhos da raiz), o nível 2 tem no máximo 4 nodos e assim por diante (veja a Figura 2.25). Em geral, o nível d em no máximo 2^d nodos, o que implica o seguinte teorema (cuja prova é deixada para o Exercício R-2.4).

Teorema 2.8: *seja T uma árvore binária (própria) com n nodos, e façamos h denotar a altura de T. Então T tem as seguintes propriedades:*

1. *o número de nodos externos de T é no mínimo* $h + 1$ *e no máximo* 2^h;
2. *o número de nodos internos de T é no mínimo h e no máximo* $2^h - 1$;
3. *o número total de nodos de T é no mínimo* $2h + 1$ *e no máximo* $2^{h+1} - 1$;
4. *a altura de T é no mínimo* $\log(n + 1) - 1$ *e no máximo* $(n - 1)/2$, *ou seja*, $\log(n + 1) - 1 \leq h \leq (n - 1)/2$.

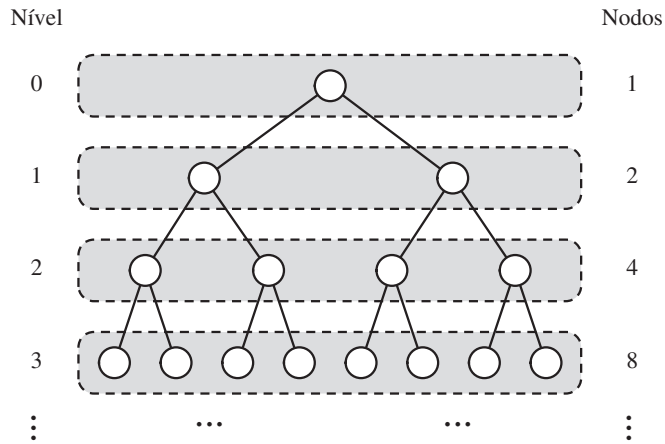


Figura 2.25 Número máximo de nodos nos níveis de uma árvore binária.

Além disso, também temos o que segue.

Teorema 2.9: *em uma árvore binária (própria) T, o número de nodos externos é uma unidade maior do que o número de nodos internos.*

Prova: a prova é por indução. Se a própria árvore T tem apenas um nodo v, então v é externo e a proposição se aplica claramente. Em qualquer outro caso, removemos de T um nodo externo (arbitrário) w e seu pai v, que é um nodo interno. Se v tem um pai u, então reconectamos u com o primeiro irmão z de w, como mostrado na Figura 2.26. Esta operação, que chamamos de removeAboveExternal(w), remove um nodo interno e um nodo externo e deixa a árvore sendo uma árvore binária própria. Logo, pela hipótese indutiva, o número de nodos externos nesta árvore é uma unidade maior do que o número de nodos internos. Chega-se a essa conclusão porque, removendo um nodo interno e um nodo externo para reduzir T uma árvore menor, a mesma propriedade continua valendo para T. ■

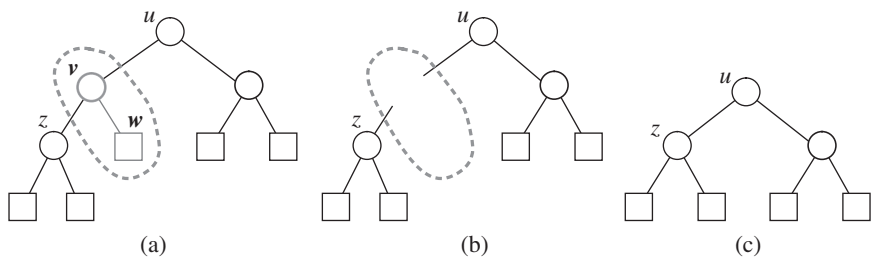


Figura 2.26 Operação removeAboveExternal(w), que remove um nodo externo e seu nodo pai, usada na justificativa do Teorema 2.9.

Observamos que a relação anterior não se aplica, em geral, para árvores que não sejam binárias. Nos capítulos seguintes, exploraremos algumas aplicações importantes desses fatos. Antes de poder discutir tais aplicações, entretanto, devemos primeiramente entender mais sobre como árvores binárias são percorridas e representadas.

Caminhamentos em árvores binárias

Como no caso de árvores genéricas, os cálculos executados em árvores binárias normalmente envolvem caminhamentos. Nesta seção, apresentamos algoritmos de caminhamento sobre árvores binárias expressos usando métodos do TAD árvore binária. Para determinar os tempos de execução, além dos pressupostos sobre o tempo de execução dos métodos do TAD árvore feitos na Seção 2.3.2, suporemos que, para uma árvore binária, o método children(v) consuma tempo $O(1)$, porque cada nodo tem ou zero ou dois filhos. Da mesma forma, supomos que os métodos leftChild(v) e rightChild(v) e sibling(v) consomem cada um tempo $O(1)$.

Caminhamento prefixado em uma árvore binária

Uma vez que qualquer árvore binária pode ser vista como uma árvore genérica, o caminhamento prefixado para árvores genéricas (Trecho de Código 2.20) pode ser aplicado a qualquer árvore binária. Podemos simplificar o pseudocódigo no caso do caminhamento sobre uma árvore binária, entretanto, como mostrado no Algoritmo 2.27.

Algoritmo binaryPreorder(T, v):
  execute a ação de "visita" para o nodo v
  se v é um nodo interno **então**
    binaryPreorder(T,T.leftChild(v)) {percorre recursivamente a subárvore esquerda}
    binaryPreorder(T,T.rightChild(v)) {percorre recursivamente a subárvore direita}

Algoritmo 2.27 Algoritmo binaryPreorder que executa o caminhamento prefixado sobre a subárvore enraizada no nodo v da árvore binária T.

Caminhamento pós-fixado em uma árvore binária

De forma análoga, o caminhamento pós-fixado para árvores genéricas (Algoritmo 2.22) pode ser especializado para árvores binárias, como pode ser visto no Algoritmo 2.28.

Algoritmo binaryPostorder(T,v):
  se v é um nodo interno **então**
    binaryPreorder(T,T.leftChild(v)) {percorre recursivamente a subárvore esquerda}
    binaryPreorder(T,T.rightChild(v)) {percorre recursivamente a subárvore direita}
  execute a ação de "visita" para o nodo v

Algoritmo 2.28 Algoritmo binaryPostorder que executa o caminhamento pós-fixado sobre a subárvore enraizada no nodo v da árvore binária T.

É interessante que a especialização dos métodos genéricos de caminhamento prefixado e pós-fixado para árvores binárias sugere um terceiro caminhamento em uma árvore binária que é diferente destes dois.

Caminhamento interfixado em árvores binárias

Um método adicional de caminhamento para árvores binárias é o caminhamento ***interfixado***. Neste caminhamento, visitamos um nodo entre os caminhamentos recursivos dos filhos da esquerda e da direita, como mostrado no Algoritmo 2.29.

Algoritmo inorder(T,v):
  se v é um nodo interno **então**
    inorder(T,T.leftChild(v)) {percorre recursivamente a subárvore esquerda}
  executa a ação de "visita" sobre o nodo v
  se v é um nodo interno **então**
    inorder(T,T.rigthChild(v)) {percorre recursivamente a subárvore direita}

Algoritmo 2.29 Algoritmo inorder para executar o caminhamento interfixado em uma subárvore de uma árvore binária T enraizada no nodo v.

O caminhamento interfixado sobre uma árvore binária T pode ser informalmente visto como uma visita aos nodos de T "da esquerda para a direita". Na verdade, para cada nodo v, o caminhamento interfixado visita v após visitar todos os nodos da subárvore esquerda de v e antes de visitar todos os nodos da subárvore direita de v (veja a Figura 2.30).

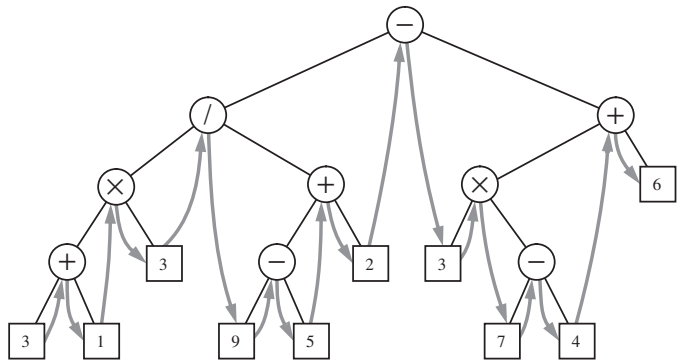


Figura 2.30 Caminhamento interfixado sobre uma árvore binária.

Um *framework* unificado para caminhamento em árvore

Os algoritmos de caminhamento em árvores que discutimos até agora são todos formas de iteradores. Cada caminhamento visita os nodos de uma árvore em uma certa ordem, e é garantido que todos os nodos são visitados exatamente uma vez. Podemos unificar os três algoritmos descritos em um único padrão de projeto; entretanto, relaxaremos o requisito de que cada nodo só pode ser visitado exatamente uma vez. O método de caminhamento resultante é chamado de ***caminhamento de Euler***, que será estudado em seguida. A vantagem desse caminhamento é que ele permite que os mais variados tipos de algoritmos sejam expressos com facilidade.

O caminhamento de Euler sobre árvores binárias

O caminhamento de Euler sobre uma árvore binária T pode ser informalmente definido como um "passeio" ao redor de T, onde iniciamos na raiz em direção ao filho da esquerda, considerando as arestas de T como "paredes" que devemos sempre manter à nossa esquerda (veja a Figura 2.3.1). Cada nodo v de T é encontrado três vezes pelo caminhamento de Euler:

- "na esquerda" (antes de percorrer a subárvore esquerda de v);
- "de baixo" (entre as duas subárvores de v);
- "na direita" (após percorrer a subárvore direita de v).

S e v for externo, então essas três visitas na verdade acontecem ao mesmo tempo.

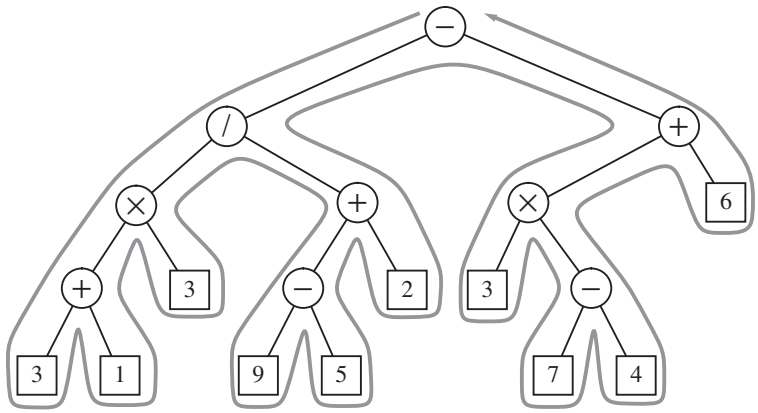


Figura 2.31 Caminhamento de Euler sobre uma árvore binária.

Fornecemos o pseudocódigo para o caminhamento de Euler sobre uma subárvore enraizada no nodo v no Algoritmo 2.32.

Algoritmo eulerTour(T,v):
  execute a ação de visita para o nodo v quando vindo da esquerda
  se v é um nodo interno **então**
    percorre recursivamente a subárvore esquerda de v chamando
    eulerTour(T,T.leftChild(v))
  execute a ação de visita para o nodo v quando vindo de baixo
  se v é um nodo interno **então**
    percorre recursivamente a subárvore direita de v chamando
    eulerTour(T,T.rightChild(v))
  execute a "ação" de visita para o nodo v quando vindo da direita

Algoritmo 2.32 Algoritmo eulerTour para computação do caminhamento de Euler da subárvore da árvore binária T enraizada de nodo v.

O caminhamento prefixado em uma árvore binária é equivalente ao caminhamento de Euler quando estão previstas ações "apenas quando vindo da esquerda". Da mesma forma, os caminhamentos interfixado e pós-fixado sobre uma árvore binária são equivalentes ao de Euler, pois cada nodo tem uma ação de "visita" associada que ocorre quando é encontrado vindo de baixo ou da direita, respectivamente.

O caminhamento de Euler estende os caminhamentos prefixados, interfixados e pós-fixados, mas também pode executar outros tipos de caminhamento. Por exemplo, suponha que desejamos calcular o número de descendentes de cada nodo v de uma árvore binária T. Iniciamos o caminhamento de Euler inicializando o contador para 0, e então incrementamos o contador cada vez que visitamos um nodo pela esquerda. Para determinar o número de descendentes de um nodo v, calculamos a diferença entre os valores do contador quando v é visitado pela esquerda e quando é visitado pelo direita, e somamos 1. Essa regra simples nos fornece o número de descendentes de v, por-

que cada nodo na subárvore enraizada em v é contado entre visitarmos v pela esquerda e pela direita. Entretanto, temos um método que consome tempo $O(n)$ para calcular o número de descendentes de cada nodo em T.

O tempo de execução do caminhamento de Euler é fácil de analisar supondo que a "visita" a um nodo consome tempo $O(1)$. Na verdade, em cada caminhamento, gastamos um tempo constante em cada nodo da árvore; logo, o tempo de execução total é $O(n)$ para um árvore com n nodos.

Outra aplicação para o caminhamento de Euler é imprimir uma expressão aritmética completamente entre parênteses a partir da árvore que a representa (Exemplo 2.4). O método printExpression, apresentado no Algoritmo 2.33, efetua essa tarefa pela execução das ações listadas a seguir durante o caminhamento de Euler:

- ação "da esquerda": se o nodo for interno, imprima "(";
- ação "de baixo": imprime o valor ou operador armazenado no nodo;
- ação "da direita": se o nodo for interno, imprima ")".

```
Algoritmo printExpression(T,v):
  se T.isExternal(v) então
    imprime o valor armazenado em v
  senão
    imprime "("
    printExpression(T,T.leftChild(v))
    imprime o operador expresso em v
    printExpression(T,T.rigthChild(v))
    imprime ")"
```

Algoritmo 2.33 Um algoritmo para imprimir a expressão aritmética associada com a subárvore de T, enraizada em v, que representa uma expressão aritmética.

Tendo presente esses exemplos em pseudocódigo, descrevemos agora diferentes maneiras eficientes de implementar o tipo abstrato de dados árvore usando estruturas concretas, tais como seqüências e listas encadeadas.

2.3.4 Estruturas de dados para representar árvores

Nesta seção, descrevemos estruturas de dados concretas para representação de árvores.

Uma estrutura baseada em vetores para árvores binárias

Uma estrutura simples para se representar uma árvore binária T é baseada em uma forma de numerar os nodos de T. Para cada nodo v de T, façamos $p(v)$ ser um inteiro definido como segue:

- se v for a raiz de T, então $p(v) = 1$;
- se v for o filho da esquerda do nodo u, então $p(v) = 2p(u)$;
- se v for o filho da direita do nodo u, então $p(v) = 2p(u) + 1$.

A função de numeração p é conhecida como ***numeradora por nível*** dos nodos de uma árvore binária T, na medida em que numera os nodos de cada nível de T em ordem crescente, da esquerda para a direita, embora possa pular alguns números (veja a Figura 2.34).

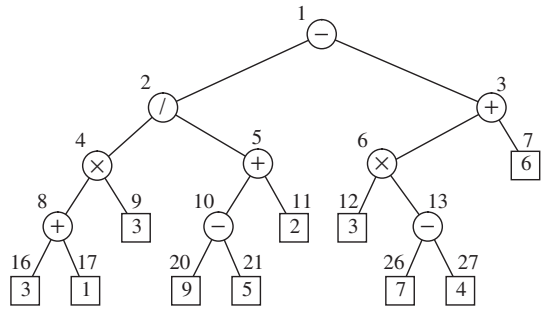


Figura 2.34 Numeração por níveis de uma árvore binária: (a) esquema geral; (b) um exemplo.

A função numeradora por nível p sugere uma representação para uma árvore binária T através de uma vetor S onde cada nodo v é associado com um elemento de S em uma colocação $p(v)$ (veja a Figura 2.35). Em geral, implementa-se o vetor S usando-se um arranjo de tamanho variável (veja a Seção 1.5.2). Tal implementação é simples e eficiente, pois nos permite executar os métodos root, parent, leftChild, rightChild, sibling, isInternal, isExternal e isRoot usando apenas operações aritméticas simples sobre os números $p(v)$ associados com cada nodo v envolvido na operação. Isto é, cada objeto posição v apenas "empacota" o índice $p(v)$ no vetor S. Deixamos os detalhes de tal implementação como um exercício (R-2.7).

Façamos n ser o número de nodos de T, e p_M ser o valor máximo de $p(v)$ considerando todos os nodos de T. O vetor S tem tamanho $N = p_M + 1$, uma vez que o elemento de S na colocação 0 não é associado com nenhum nodo de T. Além disso, o vetor S terá, normalmente, uma certa quantidade de elementos vazios que não se referem a nenhum dos nodos existentes de T. Essas posições vazias podem, por exemplo, corresponder a nodos externos vazios ou mesmo posições onde os descendentes de tais nodos estariam. Na verdade, no pior caso, $N = 2^{(n+1)/2}$, a justificativa deste fato deixamos como exercício (R-2.6). Na Seção 2.4.3, veremos uma classe de árvores binárias chamadas *heaps*, para as quais $N = n + 1$. Além do mais, se todos os nodos externos estiverem vazios, como será o caso na implementação do *heap*, poderemos salvar espaço adicional não estendendo o tamanho do vetor S para incluir nodos externos cujo índi-

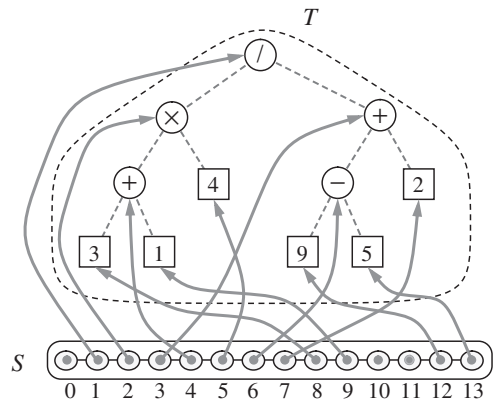


Figura 2.35 Representação de uma árvore binária T em termos de um vetor S.

ce for maior que o último nodo interno da árvore. Sendo assim, em vez do pior caso de consumo, existirão aplicações onde a representação por vetor de uma árvore binária será eficiente em termos de espaço. Porém, se considerarmos árvores binárias genéricas, o custo exponencial do pior caso de necessidade de espaço desta representação é proibitivo.

A Tabela 2.36 resume os tempos de execução dos métodos de uma árvore binária implementada sobre um vetor. Não incluímos nesta tabela nenhum método para atualizar uma árvore binária.

Operação	Tempo
positions, elements	$O(n)$
swapElements, replaceElement	$O(1)$
root, parent, children	$O(1)$
leftChild, rightChild, sibling	$O(1)$
isInternal, isExternal, isRoot	$O(1)$

Tabela 2.36 Tempos de execução dos métodos de uma árvore binária T implementada usando um vetor S, onde S é implementado usando um arranjo. Denotamos n o número de nodos de T e N o tamanho de S. Os métodos hasNext(), nextObject() e nextPosition() dos iteradores elements(), positions() e children(v) consomem tempo $O(1)$. O consumo de espaço é $O(N)$ e corresponde a $O(2^{(n+1)/2})$, no pior caso.

A implementação por vetor de uma árvore binária é uma forma rápida e fácil de implementar o TAD árvore binária, mas pode ser muito ineficiente em termos de consumo de espaço se a altura da árvore for grande. A próxima estrutura de dados que iremos apresentar para representar árvores binárias não apresenta esse problema.

Uma estrutura encadeada para árvores binárias

Uma forma natural de implementar um árvore binária T é usar uma ***estrutura encadeada***. Nesta abordagem, representamos cada nodo v de T por um objeto que referencia o elemento armazenado em v, bem como os objetos posição associados com os pais e filhos de v. Apresentamos a representação por estrutura encadeada de uma árvore binária na Figura 2.7.

Se v é a raiz de T, então a referência ao nodo pai é nula, e se v é um nodo externo, então as referências para os filhos de v são nulas.

Se quisermos guardar espaço nos casos em que os nodos externos são vazios, então tornamos as referências para os nodos externos vazios nulas, isto é, permitimos que uma referência de um nodo interno para um nodo externo filho seja nula.

Além disso, é mais direta a implementação dos métodos size(), isEmpty(), swapElements(v,w) e replaceElements(v,e) consumindo tempo $O(1)$. Além disso, o método positions() pode ser implementado executando-se um caminhamento interfixado, e a implementação do método elements() é similar. Logo, os métodos position() e elements() consomem tempo $O(n)$ cada.

Considerando o espaço requerido por essa estrutura de dados, observamos que existem objetos de tamanho constante para cada nodo da árvore T. Logo, a necessidade de espaço é $O(n)$.

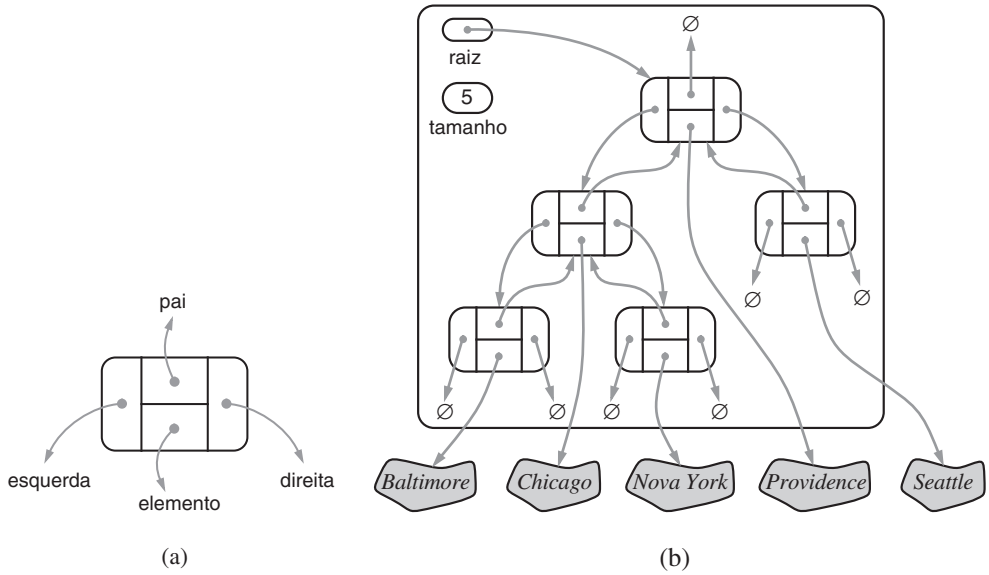


Figura 2.37 Um exemplo de estrutura de dados encadeada para representar uma árvore binária: (a) objeto associado com um nodo; (b) estrutura de dados completa com cinco nodos.

Estrutura encadeada para árvores genéricas

Podemos estender a estrutura encadeada de árvores binárias para representar árvores genéricas. Uma vez que não existe limite no número de filhos que um nodo v de uma árvore genérica pode ter, usamos um contêiner (por exemplo, uma lista ou vetor) para armazenar os filhos de v, em vez de usar variáveis de instância. Esta estrutura é apresentada de forma esquemática na Figura 2.38, assumindo que implementamos o contêiner de um nodo como uma seqüência.

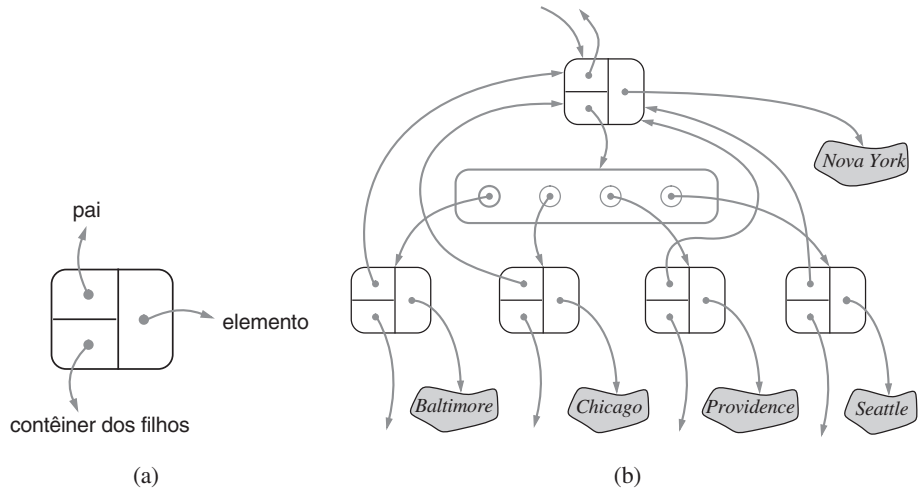


Figura 2.38 Estrutura encadeada para uma árvore: (a) o objeto associado com o nodo; (b) porção da estrutura de dados associada com um nodo e seus filhos.

Observamos que o desempenho da implementação encadeada do TAD árvore, apresentada na Tabela 2.39, é semelhante da implementação encadeada da árvore binária. A principal diferença é que, na implementação do TAD árvore, usamos um contêiner eficiente, tal como uma lista ou vetor, para armazenar os nodos de v, em vez de *links* diretos para exatamente dois filhos.

Operação	Tempo
size, isEmpty	$O(1)$
positions, elements	$O(n)$
swapElements, replaceElement	$O(1)$
root, parent	$O(1)$
children(v)	$O(c_v)$
isInternal, isExternal, isRoot	$O(1)$

Tabela 2.39 Tempos de execução para os métodos de uma árvore com n nodos implementada usando-se uma estrutura encadeada. O número de filhos de um nodo v é denotado por c_v.

2.4 Filas de prioridades e *heaps*

Nesta seção, fornecemos uma estrutura para estudar filas de prioridades baseada nos conceitos de chave e comparação.

2.4.1 O tipo abstrato de dados fila de prioridade

As aplicações comumente requerem que comparemos e classifiquemos objetos de acordo com parâmetros ou propriedades, chamadas "chaves", que são associadas a cada objeto em uma coleção. Formalmente, definimos uma ***chave*** como um objeto que é associado a um elemento como um atributo específico deste elemento, que podemos usar para identificar, classificar ou ponderar esse elemento. Note que a chave é associada a um elemento, tipicamente por um usuário ou aplicação; por isso, a chave pode representar uma propriedade que um objeto não possui originalmente.

O conceito de chave como um tipo arbitrário de objeto é, portanto, bastante genérico, mas para trabalhar consistentemente com uma definição tão geral para chaves e ainda ser capaz de discutir quando uma chave tem maior prioridade do que outra, precisamos de uma forma robusta de comparar chaves. Ou seja, uma fila de prioridade precisa de uma regra de comparação que nunca se contradiga. Para que uma regra de comparação (que denotaremos com $\leq$) seja robusta, ela deve definir uma relação de ***ordem total***, o que significa dizer que a regra de comparação é definida para cada par de chaves e deve satisfazer as seguintes propriedades:

- **Propriedade reflexiva**: $k \leq k$.
- **Propriedade anti-simétrica**: se $k_1 \leq k_2$ e $k_2 \leq k_1$, então $k_1 = k_2$.
- **Propriedade transitiva**: se $k_1 \leq k_2$ e $k_2 \leq k_3$, então $k_1 \leq k_3$.

Qualquer regra $\leq$ de comparação que satisfaça a essas três propriedades nunca levará a uma contradição nas comparações. De fato, uma regra assim define uma relação de ordem linear sobre um conjunto de chaves. Assim, se uma coleção (finita) de elementos

tem uma ordem total definida para si, então a noção de uma chave k_{min} ***menor*** é bem definida. Na verdade é a chave para a qual $k_{min} \leq k$ para qualquer outra chave k na coleção.

Uma ***fila de prioridade*** é um contêiner de elementos, cada um tendo uma chave associada atribuída no instante em que o elemento é inserido. O nome "fila de prioridade" vem do fato de que as chaves fornecem a "prioridade" usada para escolher elementos a serem removidos. Os dois métodos fundamentais de uma fila de prioridade P são:

insertItem(k,e): insere o elemento e com chave k em P.

removeMin( ): retorna e remove de P o elemento com a menor chave, ou seja, um elemento cuja chave é menor ou igual à chave de qualquer outro elemento em P.

Às vezes as pessoas se referem ao método removeMin como o método "extractMin" para enfatizar que esse método simultaneamente remove e retorna o elemento com a menor chave em P. Além disso, podemos acrescentar, a esses dois métodos fundamentais, métodos de suporte, tais como size() e isEmpty(). Podemos também acrescentar métodos de acesso, tais como os que seguem:

minElement( ): retorna (mas não remove) o elemento de P com a menor chave.

minKey( ): retorna (mas não remove) a menor chave de P.

Ambos os métodos retornam condições de erro se a fila de prioridade estiver vazia.

Um dos aspectos interessantes do TAD fila de prioridades, que agora parece óbvio, é que ele é muito mais simples que o TAD seqüência. Essa simplicidade se deve ao fato de que os elementos em uma fila de prioridades são inseridos e removidos em uma seqüência baseada em suas posições ou colocações.

Comparadores

O TAD fila de prioridades se utiliza implicitamente de um padrão de projeto: o comparador. Esse padrão especifica a forma pela qual comparamos chaves, e é projetado para suportar os tipos mais genéricos e reusáveis de filas de prioridade. Neste tipo de projeto, não devemos esperar que as chaves forneçam seu próprio mecanismo de comparação, pois tais mecanismos podem não corresponder ao desejo do usuário. Em vez disso, usamos objetos ***comparadores*** especiais, que são externos às chaves e fornecem as regras de comparação. Um comparador é um objeto que compara duas chaves. Assumimos que uma fila de prioridade P recebe um comparador quando é construída e podemos imaginar que uma fila de prioridade receba outro comparador se o antigo ficar "desatualizado". Quando P precisa comparar duas chaves, ela usa o comparador para fazer as comparações. Assim, um programador pode escrever uma implementação genérica para uma fila de prioridade que funcione corretamente em uma variedade de contextos. Formalmente, um objeto comparador provê os métodos a seguir, cada um dos quais recebendo duas chaves e as comparando (ou acusando um erro se as chaves não são comparáveis). Os métodos do TAD comparador são os seguintes:

isLess(a,b): verdadeiro se e somente se a for menor do que b.

isLessOrEqualTo(a,b): verdadeiro se e somente se a for menor ou igual a b.

isEqualTo(a,b): verdadeiro se e somente se a e b forem iguais.

isGreater(a,b): verdadeiro se e somente se a for maior do que b.

isGreaterOrEqualTo(a,b): verdadeiro se e somente se a for maior ou igual a b.

isComparable(a): verdadeiro se e somente se a puder ser comparado.

2.4.2 *PQ-Sort*, *selection-sort* e *insertion-sort*

Nesta seção, vamos discutir como usar uma fila de prioridades para ordenar um conjunto de elementos.

Ordenação *PQ-Sort*: usando uma fila de prioridades para ordenar

No problema da ordenação, temos uma coleção C de n elementos que podem ser comparados com uma relação de ordem total e desejamos reorganizá-los em ordem crescente (ou em ordem não-decrescente, se houver empates). O algoritmo para ordenar C com uma fila de prioridade Q é bastante simples e consiste nas duas fases a seguir:

1. Na primeira fase, colocamos os elementos de C em uma fila de prioridade P inicialmente vazia através de uma série de operações insertItem, uma para cada elemento.
2. Na segunda fase, retiramos os elementos de P em ordem não-decrescente através de n operações removeMin, colocando-os novamente em C em ordem.

Fornecemos o pseudocódigo deste algoritmo no Algoritmo 2.40, assumindo que C é uma seqüência (como uma lista ou vetor). O algoritmo funciona corretamente para qualquer fila de prioridade P, não interessando como P é implementada. Entretanto, o tempo de execução do algoritmo é determinado pelos tempos de execução das operações insertItem e removeMin, que dependem de como P é implementada. Assim, PriorityQueueSort deve ser considerada mais como um "esquema" de ordenação do que um algoritmo, porque não especifica como P deve ser implementada. O esquema PriorityQueueSort é o paradigma de vários algoritmos populares de ordenação, incluindo *selection-sort*, *insertion-sort* e *heapsort*, que discutiremos no restante desta seção.

Algoritmo PQ-Sort(C,P):
  Entrada: uma seqüência C com n elementos e uma fila de prioridade P que compara chaves, que são elementos de C, usando a mesma relação de ordem total.
  Saída: a seqüência C ordenada pela relação de ordem total.
  enquanto C não estiver vazio **faça**
    $e \leftarrow C$.removeFirst() {remove um elemento e de C}
    P.insertItem(e,e) {a chave é o próprio elemento}
  enquanto P não estiver vazio **faça**
    $e \leftarrow P$.removeMin() {remove o menor elemento de P}
    C.insertLast(e) {adiciona o elemento no fim de C}

Algoritmo 2.40 Algoritmo PQ-Sort. Note que os elementos da seqüência de entrada C servem tanto como chaves quanto como elementos da fila de prioridade P.

Usando uma fila de prioridades implementada com uma seqüência não-ordenada

Como em nossa primeira implementação de uma fila de prioridade P, vamos considerar armazenar os elementos de P e suas chaves em uma seqüência S. Para manter a generalidade, digamos que S é uma seqüência genérica implementada como arranjo ou lista duplamente encadeada (a escolha de uma implementação específica não irá afe-

tar o desempenho, como veremos). Assim, os elementos de S são pares (k, e), onde e é um elemento de P, e k é sua chave. Uma maneira simples de implementar o método insertItem(k,e) de P é adicionar o novo par $p = (k, e)$ no final da seqüência S executando o método insertLast(p) em S. Essa implementação de insertItem tem tempo $O(1)$ sem importar se a seqüência é implementada com arranjos ou listas encadeadas (veja a Seção 2.2.3). Essa escolha implica que S não será ordenada, pois a inserção sempre no final de S não leva em conta a ordem das chaves. Como conseqüência, para realizar a operação minElement, minKey ou removeMin em P, devemos inspecionar todos os elementos de S para achar o elemento $p = (k, e)$ com o menor valor de k. Assim, independentemente de como a seqüência S é implementada, esses métodos de procura em P sempre custam tempo $O(n)$, onde n é o número de elementos em P quando o método é executado. Além disso, esses métodos são executados em tempo $\Omega(n)$ mesmo no melhor caso, pois cada um deles requer que toda a seqüência seja pesquisada para se encontrar o menor elemento. Ou seja, estes métodos são executados em tempo $\Omega(n)$. Desta forma, usando uma seqüência não-ordenada para implementar uma fila de prioridade, obtemos inserção em tempo constante, sendo que a operação removeMin exige tempo linear.

Selection-sort

Se implementamos a fila de prioridade P com uma seqüência não-ordenada, então a primeira fase do PQ-Sort custa tempo $O(n)$, pois podemos inserir cada elemento em tempo constante. Na segunda fase, assumindo que podemos comparar duas chaves em tempo constante, o tempo de execução de cada operação removeMin é proporcional ao número de elementos em P. Assim, o gargalo dessa implementação é a repetida "seleção" do elemento mínimo de uma seqüência não-ordenada na fase 2. Por este motivo, este algoritmo é mais conhecido como ***selection-sort*** (veja a Figura 7.1).

Vamos analisar o algoritmo *selection-sort*. Como mostrado acima, o gargalo é a segunda fase, na qual repetidamente removemos o elemento com menor chave da fila de prioridade P. O tamanho de P começa em n e diminui em um a cada operação removeMin até chegar a zero. Assim, a primeira operação removeMin custa tempo $O(n)$, a segunda custa tempo $O(n-1)$ e assim por diante, até que a última (n-ésima) operação custa tempo $O(1)$. Portanto, o tempo total exigido pela segunda fase é

$$O(n + (n-1) + \cdots + 2 + 1) = O\left(\sum_{i=1}^{n} i\right).$$

Pelo Teorema 1.13, temos $\sum_{i=1}^{n} = \frac{n(n+1)}{2}$. Assim, a segunda fase custa tempo $O(n^2)$, como também exige o algoritmo completo.

Usando uma fila de prioridades com uma seqüência ordenada

Uma implementação alternativa para uma fila de prioridade P também usa uma seqüência S, mas desta vez os elementos são armazenados em ordem de chave. Podemos implementar os métodos minElement e minKey neste caso, simplesmente acessando o primeiro elemento da seqüência usando o método first de S. Da mesma forma, podemos implementar o método removeMin de P como sendo S.remove(S.first()). Assumindo que S é implementada como uma lista encadeada ou um arranjo que suporta remoção do primeiro elemento em tempo constante (veja a Seção 2.2.3), achar e remover o mínimo em P requer tempo $O(1)$. Assim, usar uma seqüência ordenada permite uma

implementação simples e rápida para acesso a uma fila de prioridade e para os métodos de remoção.

Esse benefício tem um custo, no entanto, pois agora o método insertItem de P requer que vasculhemos a seqüência S para determinar a posição apropriada para inserir o novo elemento e sua chave. Assim, implementar o método insertItem de P agora exige tempo $O(n)$, onde n é o número de elementos em P no momento em que o método é executado. Em suma, quando usamos uma seqüência ordenada para implementar uma fila de prioridade, a inserção é executada em tempo linear enquanto a busca e a remoção de mínimos podem ser feitas em tempo constante.

Insertion-sort

Se implementamos a fila de prioridade P com uma seqüência ordenada, podemos melhorar o tempo de execução da segunda fase para $O(n)$, pois cada operação removeMin em P custa tempo $O(1)$. Infelizmente, neste caso, a primeira fase se torna o gargalo para o tempo de execução. De fato, no pior caso, o tempo de execução de cada operação insertItem é proporcional ao número de elementos que se encontram na fila de prioridade, que começa com tamanho zero e aumenta até ter tamanho n. A primeira operação insertItem custa tempo $O(1)$, a segunda custa tempo $O(2)$ e assim por diante, até que a última (n-ésima) operação custa tempo $O(n)$ no pior caso. Assim, se usarmos uma seqüência ordenada para implementar P, então a primeira fase se torna a fase onde o gargalo acontece. Este algoritmo de ordenação é, portanto, mais conhecido como ***insertion-sort***, pois o gargalo desse algoritmo envolve a repetida "inserção" de um novo elemento na posição apropriada da seqüência ordenada.

Analisando o tempo de execução da fase 1 do *insertion-sort*, notamos que é $O(\sum_{i=1}^{n} i)$ no pior caso. Novamente, pelo Teorema 1.13, a primeira fase é executada em tempo $O(n^2)$, bem como todo o algoritmo.

Em outras palavras, o tempo de execução do esquema PriorityQueueSort implementado com uma seqüência ordenada é $O(n^2)$. Portanto, tanto o *selection-sort* quanto o *insertion-sort* têm tempos de execução $O(n^2)$.

Embora *selection-sort* e *insertion-sort* sejam similares, ele possuem algumas diferenças interessantes. Por exemplo, note que o *selection-sort* sempre toma tempo $\Omega(n^2)$, pois selecionar o mínimo a cada passo da segunda fase requer que vasculhemos toda a seqüência da fila de prioridade. O tempo de execução do *insertion-sort*, por outro lado, depende da seqüência dos dados de entrada. Por exemplo, se os dados estão em ordem reversa, então o *insertion-sort* é executado em tempo $O(n)$.

2.4.3 A estrutura de dados *heap*

Uma implementação eficiente de uma fila de prioridade usa uma estrutura de dados chamada ***heap***. Esta estrutura de dados permite que realizemos inserções e deleções em tempo logarítmico. A maneira pela qual um *heap* obtém este avanço é abandonando a idéia de armazenar os elementos e as chaves em uma seqüência e, em vez disso, armazená-los em uma árvore binária.

Um *heap* (veja a Figura 2.41) é uma árvore binária T que armazena uma coleção de chaves em seus nodos internos e que satisfaz duas propriedades adicionais: uma propriedade relacional definida em termos da forma com que as chaves são armazenadas em T e uma propriedade estrutural definida em termos dos próprios nodos de T.

Assumimos que uma relação de ordem total é fornecida para as chaves, por exemplo, através de um comparador. Note que em nossa definição de *heap*, os nodos externos de T não armazenam chaves ou elementos, e servem apenas para ocupar os espaços vazios. A propriedade relacional de T, é a seguinte:

Propriedade de ordem do *heap*: em um *heap* T, para cada nodo v diferente da raiz, a chave em v é maior ou igual à chave armazenada no nodo pai de v.

Como uma conseqüência da propriedade de ordem do *heap*, as chaves encontradas em um caminho da raiz até um nodo externo de T estão em ordem não-decrescente. A chave com valor mínimo sempre está armazenada na raiz de T. Para aumentar a eficiência, desejamos que um *heap* T tenha a menor altura possível. Efetivamos esse desejo através de uma condição estrutural adicional:

Árvore binária completa: uma árvore binária T com altura h é ***completa*** se os níveis $0, 1, 2, \ldots, h-1$ tiverem o maior número de nodos possível (ou seja, o nível i tem 2^i nodos para $0 \leq i \leq h-1$) e, no nível $h-1$, todos os nodos internos estiverem à esquerda dos nodos externos.

Dizendo que todos os nodos internos do nível $h-1$ estão "à esquerda" dos nodos externos, dizemos que todos os nodos internos deste nível serão visitados antes dos nodos externos deste nível se usarmos um caminhamento-padrão (por exemplo, caminhamento interfixado). Ou seja, se desenharmos uma árvore binária, todos os nodos internos no nível $h-1$ são desenhados à esquerda de quaisquer nodos externos do mesmo nível (veja a Figura 2.41).

Insistindo que um *heap* T seja completo, identificamos outro nodo importante do *heap* T, diferente da raiz, chamado de o ***último nodo*** de T, que definimos como sendo o nodo mais interno e mais à direita de T (veja a Figura 2.41).

Implementando uma fila de prioridade com um *heap*

Nossa estrutura de dados baseada em *heap* para uma fila de prioridade P consiste no seguinte (veja a Figura 2.42):

- ***heap***, uma árvore binária completa T cujos elementos são armazenados em nodos internos e cujas chaves respeitam a propriedade de ordem do *heap*. Assumimos que a árvore binária T é implementada usando-se um vetor, como des-

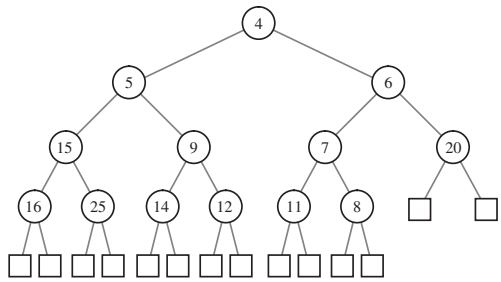


Figura 2.41 Exemplo de um *heap* armazenando 13 chaves inteiras. O último nodo é o que armazena a chave 8, e os nodos externos estão vazios.

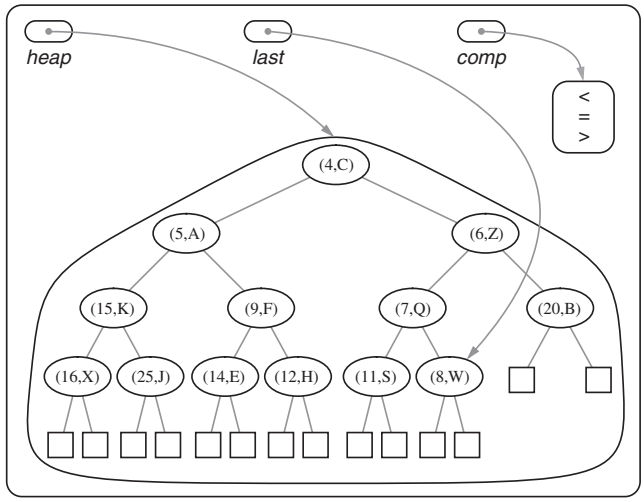


Figura 2.42 Uma fila de prioridades baseada em *heap* armazenando chaves inteiras e elementos de texto.

crito na Seção 2.3.4. Para cada nodo interno v de T, denotamos a chave do elemento armazenado em v como $k(v)$.

- ***last***, uma referência para o último nodo de T. Dada a implementação de T como um vetor, assumimos que a variável de instância ***last*** é um índice inteiro para a célula do vetor que armazena o último nodo de T.
- ***comp***, um comparador que define a relação de ordem total entre as chaves. Sem perder generalidade, assumimos que ***comp*** mantém o elemento mínimo na raiz. Se, em vez disso, desejarmos que o elemento máximo esteja na raiz, então podemos redefinir nossa regra de comparação da mesma forma.

A eficiência de tal implementação esta baseada no seguinte fato:

Teorema 2.10: *um heap T armazenando n chaves tem altura* $h = \lceil \log(n + 1) \rceil$.

Prova: já que T é completo, sabemos que o número de nodos internos é pelo menos

$$1 + 2 + 4 + \cdots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$$

Esse limite inferior é obtido quando existe apenas um nodo interno no nível $h - 1$. Alternativamente, mas também vindo do fato de que T é completo, podemos dizer que o número de nodos internos de T é no máximo

$$1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1.$$

Esse limite superior é obtido quando todos os 2^{h-1} nodos no nível $h - 1$ são internos. Já que o número de nodos internos é igual ao número n de chaves, $2^{h-1} \leq n$ e $n \leq 2^h - 1$. Assim, usando logaritmos de ambos os lados dessas desigualdades, vemos que $h \leq \log n + 1$ e $\log (n + 1) \leq h$, o que implica $h = \lceil \log(n + 1) \rceil$. ■

Logo, se executarmos operações de atualização em um *heap* em tempo proporcional à sua altura, então essas operações irão executar em tempo logarítmico.

Representação de um *heap* como um vetor

Note que, quando o *heap T* é implementado com um vetor, o índice do último nodo *w* é sempre igual a *n*, e o primeiro nodo externo (e vazio) *z* tem um índice igual a $n + 1$ (veja a Figura 2.43). Note que este índice para *z* é válido até mesmo para os seguintes casos:

- Se o último nodo corrente *w* for o nodo mais à direita do seu nível, então *z* é o nodo mais à esquerda do último nível (veja a Figura 2.43b).
- Se *T* não tiver nodos internos (ou seja, se a fila de prioridade estiver vazia e o último nodo de *T* não estiver definido), então *z* é a raiz de *T*.

A simplificação obtida pela representação do *heap T* como um vetor auxilia nossos métodos para a implementação do TAD fila de prioridade. Por exemplo, os métodos de atualização expandExternal(*z*) e removeAboveExternal(*z*) podem ser realizados em tempo $O(1)$ (presumindo que não seja necessário fazer uma expansão no vetor), pois envolvem simplesmente alocar ou desalocar uma única posição do vetor. Com essa estrutura de dados, os métodos size e isEmpty( ) custam tempo $O(1)$, como é usual. Adicionalmente, os métodos minElement e minKey também podem ser realizados facilmente em tempo $O(1)$, acessando-se o elemento ou a chave armazenada na raiz do *heap* (que está na posição 1 do arranjo). Além disso, como *T* é uma árvore binária completa, o arranjo associado ao *heap T* em uma implementação de *heaps* baseada em arranjos tem $2n + 1$ elementos, $n + 1$ dos quais são nodos externos, sem informação por nossa convenção. De fato, já que todos os nodos externos têm índices maiores do que todos os nodos internos, não precisamos armazenar explicitamente todos os nodos externos (veja a Figura 2.43).

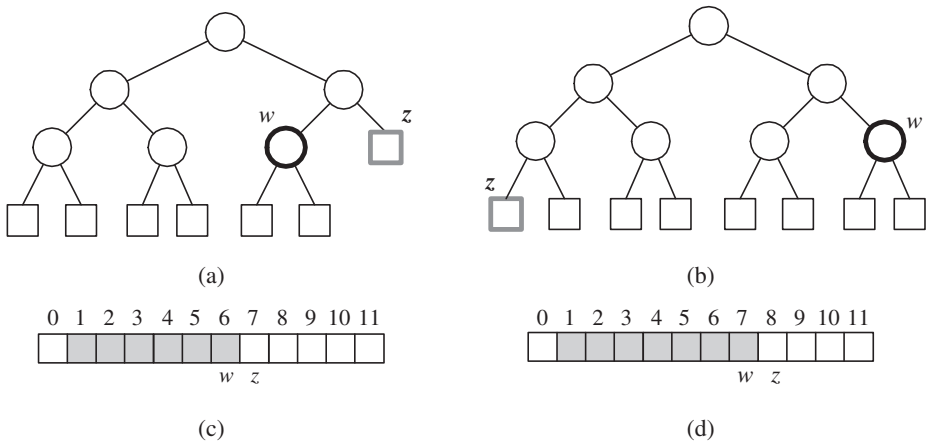


Figura 2.43 Último nodo *w* e primeiro nodo externo *z* em um *heap*: (a) caso normal onde *z* está à direita de *w*; (b) caso onde *z* está mais à esquerda no último nível. A representação vetorial de (a) é mostrada em (c); da mesma forma, a representação de (b) é apresentada em (d).

Inserção

Vamos considerar como efetuar o método insertItem do TAD fila de prioridade usando o *heap T*. Para armazenar um novo par chave-elemento (k, e) em *T*, precisamos adicionar um novo nodo interno a *T*. Para manter *T* como uma árvore binária completa, devemos adicionar este novo nodo de forma que ele seja o novo último nodo de *T*. Ou se-

ja, devemos identificar corretamente o nodo externo z onde podemos efetuar uma operação expandExternal(z) e inserir o novo elemento em z mantendo T completa (veja a Figura 2.44a-b). O nodo z é chamado de ***posição de inserção***.

Usualmente, o nodo z é o nodo externo imediatamente à direita do último nodo w (veja a Figura 2.43a). Em todo caso, pela nossa implementação de T baseada em vetores, a posição z de inserção é a posição $n + 1$ do vetor, onde n é o tamanho atual do *heap*. Assim, podemos identificar o nodo z em tempo constante no arranjo implementando T. Depois de efetuar expandExternal(z), o nodo z se torna o último nodo, e armazenamos nele o novo par chave-elemento (k, e) de forma que $k(z) = k$.

Bubbling-up após uma inserção

Após esta ação, a árvore T está completa, mas pode estar violando a propriedade de ordem do *heap*. Portanto, a não ser que o nodo z seja a raiz de T (ou seja, a fila de prioridade estava vazia antes da inserção), comparamos a chave $k(z)$ com a chave $k(u)$ armazenada em u, o nodo pai de z. Se $k(u) > k(z)$, então precisamos restaurar a propriedade de ordem, o que pode ser feito localmente trocando-se os pares de chave-elemento armazenados em u e z (veja a Figura 2.44c-d). Essa troca faz com que o novo par chave-elemento (k, e) mova-se para cima um nível. Novamente, a propriedade de ordem pode ter sido violada, e continuamos subindo o par (k, e) através de trocas sucessivas até que não haja mais uma violação da propriedade (veja a Figura 2.44e-h).

O movimento para cima através de trocas é convencionalmente chamado de ***heap bubbling-up***, ***bubbling-up*** ou simplesmente ***bubbling***. Uma troca pode resolver o problema da violação da propriedade de ordem ou mover a violação um nível acima no *heap*. No pior caso, o *bubbling* faz com que o par chave-elemento suba até a raiz do *heap* T (veja a Figura 2.44). Assim, no pior caso, o tempo de execução do método insertItem é proporcional à altura de T, ou seja, é $O(\log(n))$, pois T é completa.

Se T é implementado com um vetor, então podemos encontrar o novo último elemento z imediatamente em tempo $O(1)$. Por exemplo, poderíamos usar o mecanismo de herança para estender uma implementação de árvore binária baseada em vetor, adicionando um método que retorna o nodo com índice $n + 1$, ou seja, com o número de nível $n + 1$ como definido na Seção 6.4.1. De forma alternativa, poderíamos definir um método add que adiciona um novo elemento no primeiro nodo externo z na colocação $n + 1$ do vetor. No Algoritmo 2.60, mostrado mais tarde neste capítulo, mostramos como usar este método para implementar o método insertItem eficientemente. Se, por outro lado, o *heap* T é implementado com uma estrutura encadeada, então achar a posição z de inserção é um pouco mais complicado (veja o Exercício C-2.27).

Remoção

Vamos agora tratar do método removeMin no TAD fila de prioridade. O algoritmo para realizar o método removeMin usando o *heap* T é ilustrado na Figura 2.45.

Sabemos que um elemento com a menor chave é armazenado na raiz r de T (mesmo se existir mais de um elemento com a menor chave). Entretanto, a não ser que r seja o único nodo interno de T, não podemos simplesmente deletar o nodo r, pois essa ação iria quebrar a estrutura da árvore binária. Em vez disso, acessamos o último nodo w de T, copiamos seu par chave-elemento para a raiz r e deletamos o último nodo com uma operação removeAboveExternal(T.RightChild(w)) do TAD árvore binária. Essa operação remove w e seu filho direito, substituindo w por seu filho esquerdo (veja a Figura 2.45a-b).

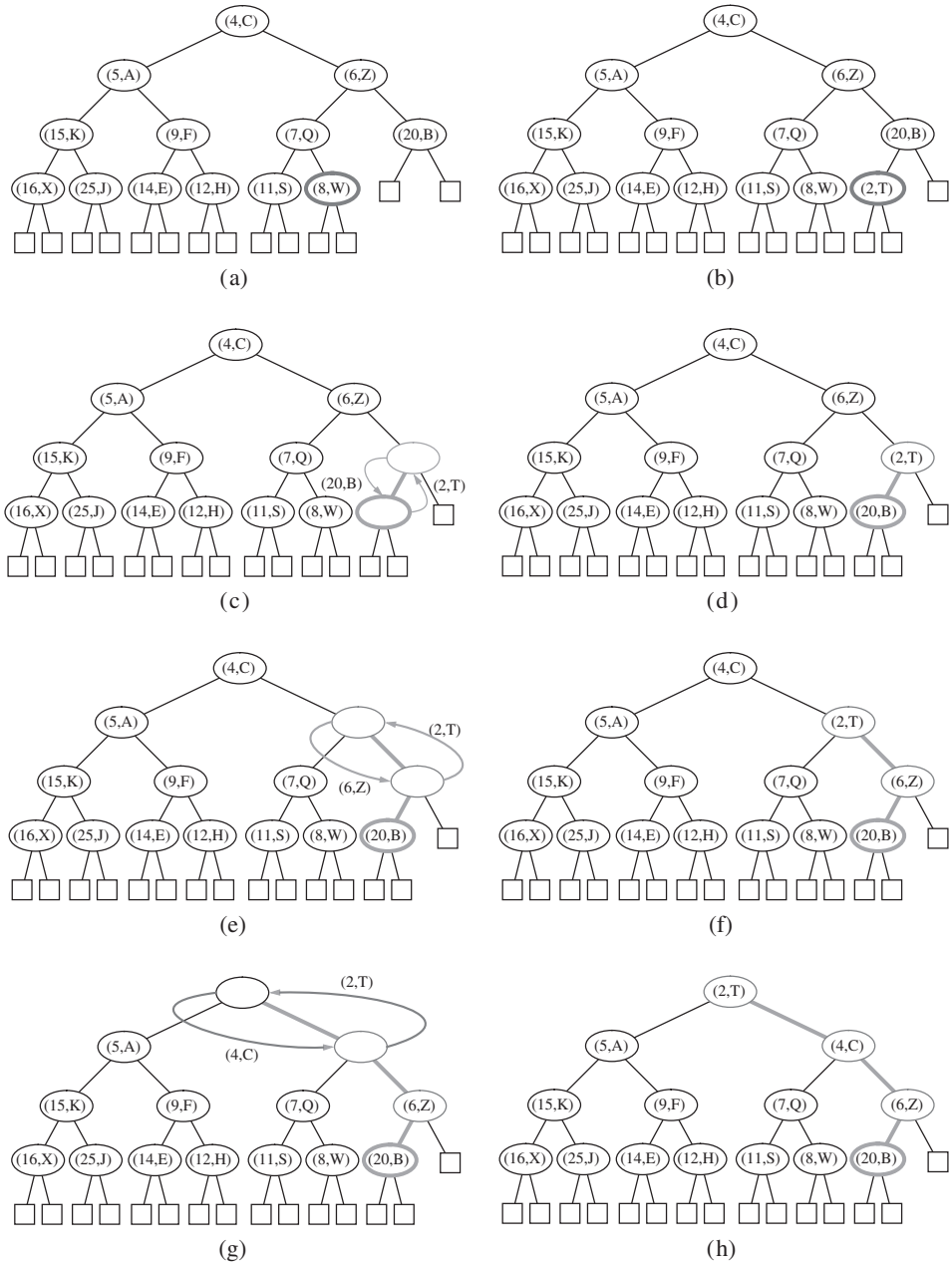


Figura 2.44 Inserção de um novo elemento com chave 2 no *heap* da Figura 2.42: (a) *heap* inicial; (b) adicionando um novo último nodo à direita do antigo último nodo; (c-d) troca para restaurar localmente a propriedade de ordem do *heap*; (e-f) outra troca; (g-h) troca final.

Após essa ação de tempo constante, precisamos atualizar nossa referência para o último nodo, o que pode ser feito simplesmente referenciando o nodo na posição n (após a remoção) no arranjo que implementa a árvore T.

Bubbling-down após uma remoção

Ainda não terminamos, entretanto, porque, mesmo que T esteja completa agora, T pode estar violando a propriedade de ordem do *heap*. Para determinar se precisamos restaurar esta propriedade, examinamos a raiz r de T. Se os filhos de r forem nodos externos, então a propriedade é trivialmente satisfeita. Senão, diferenciamos dois casos:

- Se o filho esquerdo de r for interno e o da direita for externo, então chamamos o filho esquerdo de s.
- Em outro caso (ambos os filhos de r forem internos), chamamos de s o filho de r com a menor chave.

Se a chave $k(r)$ armazenada em r for maior do que a chave $k(s)$ armazenada em s, então precisamos restaurar a propriedade de ordem do *heap*, o que pode ser localmente feito trocando-se os pares chave-elemento em r e em s (veja a Figura 2.45c-d). A troca restaura a propriedade de ordem do *heap* para o nodo r e seus filhos, mas pode violar esta propriedade em s; portanto, podemos ter de continuar fazendo trocas em T até que não aconteça mais uma violação (veja a Figura 2.45e-h).

Essas trocas descendentes são chamadas de ***down-heap bubbling***, ou simplesmente ***down-bubbling***. Uma troca resolve a violação da propriedade ou a propaga um nível para baixo no *heap*. No pior caso, um par chave-elemento move-se todo o caminho até o nível imediatamente acima do último (veja a Figura 2.45). Assim, o tempo de execução do método removeMin é, no pior caso, proporcional à altura do *heap* T, ou seja, $O(\log(n))$.

Desempenho

A Tabela 2.46 mostra o tempo de execução dos métodos do TAD fila de prioridade para a implementação baseada em *heap*, assumindo que o *heap* T é implementado por uma estrutura de dados para árvores binárias que suporta os métodos para o TAD árvore binária (exceto elements()) em tempo $O(1)$. A estrutura encadeada e a estrutura baseada em vetor da Seção 2.3.4 satisfazem facilmente esta exigência.

Resumindo, cada um dos métodos do TAD fila de prioridade pode ser feito em tempo $O(1)$ ou em tempo $O(\log(n))$, onde n é o número de elementos no momento em que um método é executado. A análise do tempo de execução dos métodos é baseada no seguinte:

- A altura do heap T é $O(\log(n))$, pois T é completo.
- No pior caso, *bubbling* e *down-bubbling* levam tempo proporcional à altura de T.
- Encontrar a posição para inserção na execução de insertItem e atualizar a posição do último nodo em removeMin levam tempo constante.
- O *heap* T tem n nodos internos, cada um armazenando uma referência para uma chave e um elemento e $n + 1$ nodos internos.

Concluímos dizendo que a estrutura de dados *heap* é uma realização muito eficiente do TAD fila de prioridade, independentemente do *heap* ser implementado através de uma estrutura encadeada ou de uma seqüência. A implementação do *heap* consegue baixos tempos de execução para inserção e remoção, diferentemente das filas de prioridade implementadas com seqüências. De fato, uma importante conseqüência da eficiência da implementação do *heap* é que ela pode acelerar o método de ordenação

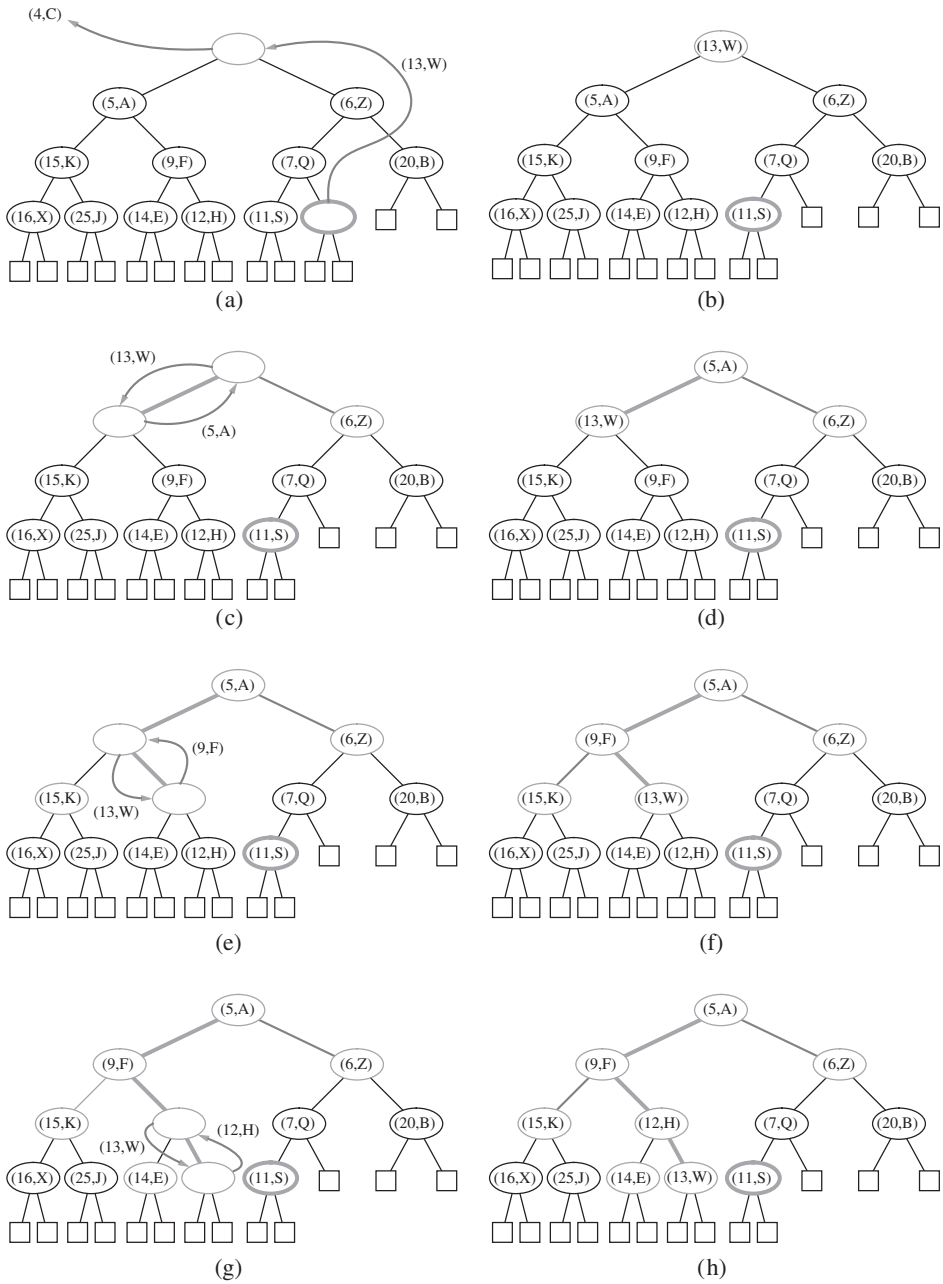


Figura 2.45 Remoção do elemento do *heap* com menor chave: (a-b) deleção do último nodo, cujo par chave-elemento é armazenado na raiz; (c-d) troca para restaurar localmente a propriedade de ordem do *heap*; (e-f) outra troca; (g-h) troca final.

Operação	Tempo
size, isEmpty	$O(1)$
minElement, minKey	$O(1)$
insertItem	$O(\log n)$
removeMin	$O(\log n)$

Tabela 2.46 Desempenho de uma fila de prioridade implementada através de um *heap*, por sua vez implementado com uma estrutura baseada em vetor para representar uma árvore binária. Indicamos com n o número de elementos na fila de prioridade no momento em que um método é executado. A necessidade de espaço é $O(n)$ se o *heap* for implementado por uma estrutura encadeada, e $O(N)$ se o *heap* for implementado com um vetor, onde $N \geq n$ é o tamanho do vetor usado para a implementação.

baseado em fila de prioridade, tornando-o muito mais veloz do que os métodos *insertion-sort* e *selection-sort*, baseados em seqüências.

2.4.4 *Heapsort*

Vamos considerar novamente o esquema de ordenação PQ-Sort visto na Seção 2.4.2, que usa uma fila de prioridades P para ordenar uma seqüência S. Se implementarmos a fila de prioridade P com um *heap*, então durante a primeira fase cada uma das n operações insertItem custa tempo $O(\log(k))$, onde k é o número de elementos no *heap* no momento da inserção. Da mesma forma, durante a segunda fase, cada uma das n operações removeItem custa tempo $O(\log(k))$, onde k é o número de elementos no *heap* no momento da remoção. Já que sempre temos $k \leq n$, cada uma dessas operações é executada em tempo $O(\log(n))$ no pior caso. Assim, cada fase leva tempo $O(n \log(n))$ quando usamos um *heap* para implementar a fila de prioridade. Esse algoritmo de ordenação é conhecido como ***heapsort***, e seu desempenho é descrito no teorema a seguir.

Teorema 2.11: *o algoritmo heapsort ordena uma seqüência S de n elementos comparáveis entre si em tempo $O(n \log(n))$.*

Revendo a Tabela 1.7, enfatizamos que o tempo de execução $O(n \log(n))$ do *heapsort* é consideravelmente melhor do que o tempo de execução $O(n^2)$ da *selection-sort* e *insertion-sort*. Além disso, é possível fazer várias modificações no algoritmo *heapsort* que, na prática, podem melhorar seu desempenho.

Implementando *heapsort in-place*

Se a seqüência S a ser ordenada é implementada através de um arranjo, podemos acelerar o *heapsort* e reduzir sua necessidade de memória em um fator constante usando uma parte da própria seqüência S para armazenar o *heap* e evitando o uso de uma estrutura de dados *heap* externa. Isso é feito modificando-se o algoritmo da forma a seguir:

1. Usamos um comparador reverso, o que corresponde a um *heap* com o maior elemento no topo. A qualquer momento durante a execução do algoritmo, usa-

mos a parte esquerda de S até uma certa posição $i - 1$ para armazenar os elementos no *heap* e a parte direita de S das posições i até $n - 1$ para armazenar os elementos da seqüência. Assim, os primeiros i elementos de S (nas posições $0,..., i - 1$) fornecem uma representação com vetor para o *heap* (com numeração iniciando em 0, e não mais em 1), ou seja, o elemento na posição k do *heap* é "maior ou igual aos filhos" nas posições $2k + 1$ e $2k + 2$.

2. Na primeira fase do algoritmo, começamos com um *heap* vazio e movemos a fronteira entre o *heap* e a seqüência da esquerda para a direita, um passo de cada vez. No passo i, ($i = 1,..., n$) expandimos o *heap* adicionando o elemento na posição $i - 1$.
3. Na segunda fase do algoritmo, iniciamos com uma seqüência vazia e movemos a fronteira entre o *heap* e a seqüência da direita para a esquerda, um passo de cada vez. No passo i ($i = 1,..., n$) removemos o maior elemento do *heap* e o armazenamos na posição $n - i$.

Esta variação do *heapsort* é dita ***in-place*** ("no local"), pois usa um espaço adicional constante além da própria seqüência. Em vez de retirar elementos da seqüência e depois recolocá-los, simplesmente os rearranjamos. Ilustramos esta versão do *heapsort* na Figura 2.47. Em geral, dizemos que um algoritmo de ordenação é *in-place* se ele usar uma quantidade constante de memória além da memória necessária para armazenar os elementos a serem ordenados. A vantagem do algoritmo *in-place* de ordenação, na prática, é que ele faz um uso mais eficiente da memória principal do computador onde está executando.

Construção *bottom-up* do *heap*

A análise do algoritmo *heapsort* mostra que podemos construir um *heap* armazenando n pares chave-elemento em tempo $O(n \log(n))$ através de n operações insertItem e depois usar o *heap* para retirar os elementos em ordem decrescente. No entanto, se todas as chaves a ser armazenadas no *heap* são dadas previamente, existe um método alternativo de construção que monta o *heap* de baixo para cima (***bottom-up***) em tempo $O(n)$.

Descreveremos esse método nesta seção, observando que ele poderia ser incluído como um dos construtores da classe Heap, em vez de construir-se um *heap* usando-se uma série de n operações insertItem. Para manter a simplicidade, descreveremos essa forma de construção presumindo que o número n de chaves é um inteiro da forma

$$n = 2^h - 1$$

Ou seja, o *heap* é uma árvore binária completa com cada nível completo; portanto, o *heap* tem altura

$$h = \log(n + 1).$$

Descrevemos a construção *bottom-up* do *heap* usando um algoritmo recursivo, como mostrado no Algoritmo 2.48, que chamamos passando a seqüência que armazena as chaves com as quais queremos construir o *heap*. Descrevemos o algoritmo de construção que atuam sobre as chaves, entendendo que os elementos correspondentes as acompanham. Assim, os itens armazenados na árvore T são pares elemento-chave.

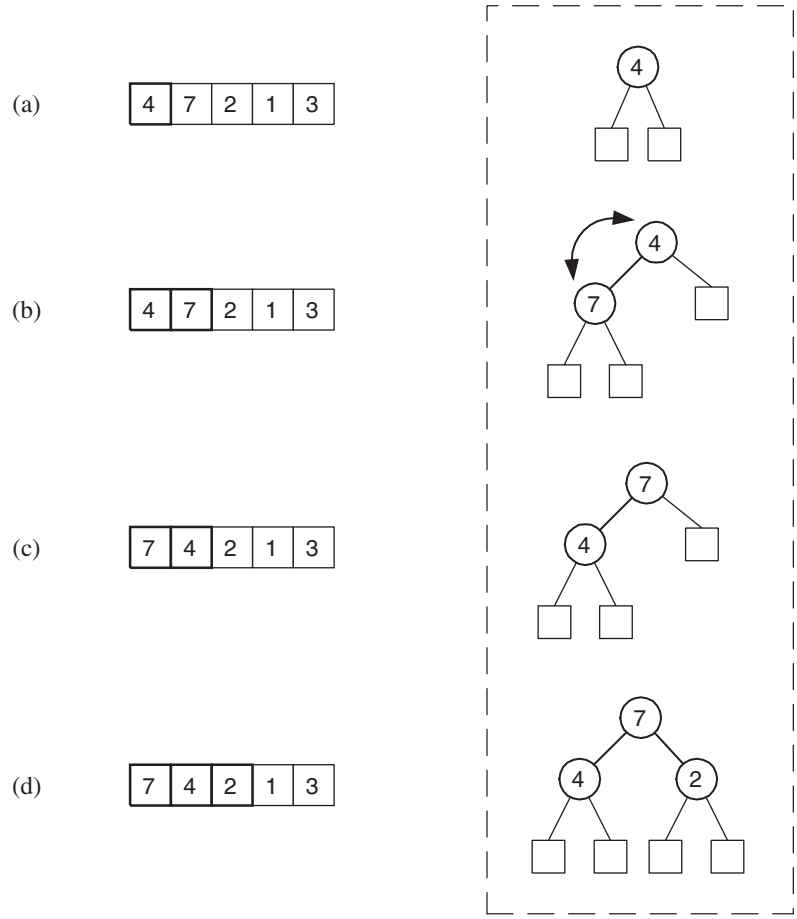


Figura 2.47 Os primeiros três passos da fase 1 do *heapsort in-place*. A parte do *heap* no arranjo é destacada usando-se uma linha grossa. Desenhamos ao lado do arranjo a árvore binária representando o *heap*, mesmo que esta árvore não seja realmente construída pelo algoritmo *in-place*.

Algoritmo BottomUpHeap(S):
Entrada: uma seqüência S que armazena $n = 2^h - 1$ chaves
Saída: um *heap* T que armazena as chaves de S.
se S for vazio **então**
  retorne um *heap* vazio (consistindo em um único nodo externo).
Remova a primeira chave k de S.
Divida S em duas seqüências S_1 e S_2, cada uma com tamanho $(n - 1)/2$.
$T_1 \leftarrow$ BottomUpHeap(S_1)
$T_2 \leftarrow$ BottomUpHeap(S_2)
Crie uma árvore binária T cuja raiz r armazena k, a subárvore esquerda T_1 e a subárvore direita T_2.
Execute um down-heap-bubbling a partir da raiz r de T, se necessário.
retorne T

Algoritmo 2.48 Construção *bottom-up* recursiva do *heap*.

Esse algoritmo de construção é chamado de construção *bottom-up* do *heap* por causa da forma pela qual as chamadas recursivas retornam uma subárvore, que é um *heap* para os elementos que armazena. Assim, a "heapificação" de T começa em seus nodos externos e continua árvore acima, à medida que as chamadas recursivas retornam. Por essa razão, alguns autores se referem à construção *bottom-up* do *heap* como uma operação de "heapificação".

Demonstramos a construção *bottom-up* de um *heap* na Figura 2.49 para $h = 4$.

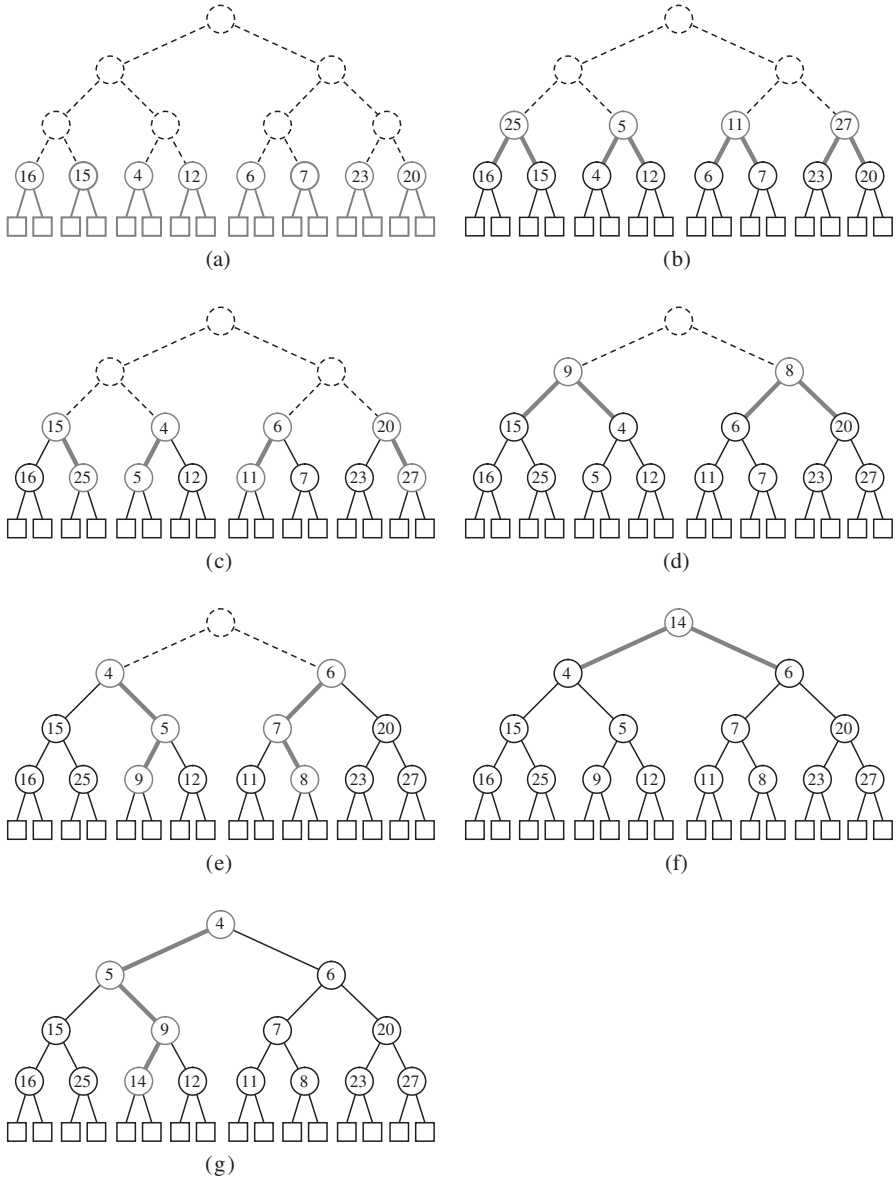


Figura 2.49 Construção *bottom-up* de um *heap* com 15 elementos: (a) começamos construindo *heaps* de uma chave no nível inferior; (b-c) combinamos esses *heaps* em *heaps* de três chaves e em (d-e) *heaps* de sete chaves até que em (f-g) completamos o *heap*. Os caminhos do processo de *down-bubbling* são mostrados em linhas grossas.

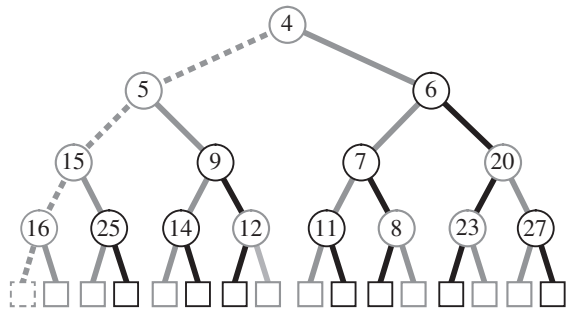


Figura 2.50 Demonstração visual do tempo linear de execução de construção *bottom-up* de um *heap* onde os caminhos associados com os nodos internos foram destacados alternando-se cinza e preto. Por exemplo, o caminho associado com a raiz consiste nos nodos internos que armazenam as chaves 4, 6, 7 e 11 mais um nodo externo.

A construção *bottom-up* é assintoticamente mais rápida do que a construção incremental através de n inserções em um *heap* vazio, como demonstra o teorema a seguir.

Teorema 2.12: *a construção bottom-up com n chaves custa tempo* $O(n)$.

Prova: analisamos a construção *bottom-up* do *heap* usando uma abordagem "visual", ilustrada na Figura 2.50.

Sendo T o *heap* final e v um nodo interno de T, identificamos como $T(v)$ a subárvore de T com raiz v. No pior caso, o tempo para formar $T(v)$ a partir das duas subárvores formadas recursivamente e tendo os filhos de v em suas raízes é proporcional à altura de $T(v)$. O pior caso acontece quando *down-bubbling*, a partir de v, atravessa um caminho de v até um dos nodos mais externos de $T(v)$. Considere agora o caminho $p(v)$ em T, do nodo v até seu sucessor externo (em um caminhamento prefixado), ou seja, o caminho que inicia em v, visitar o filho direito de v e descer sempre para a esquerda até chegar a um nodo externo. Dizemos que o caminho $p(v)$ é ***associado com*** o nodo v. Note que $p(v)$ não é necessariamente o caminho seguido em um *down-bubbling* quando $T(v)$ é formado. Claramente, o comprimento (número de arestas) de $p(v)$ é igual à altura de $T(v)$. Portanto, formar $T(v)$ leva, no pior caso, tempo proporcional ao comprimento de $p(v)$. Assim, o tempo de execução total da construção *bottom-up* é proporcional à soma dos comprimentos dos caminhos associados aos nodos internos de T.

Note que, para quaisquer nodos internos u e v de T, os caminhos $p(u)$ e $p(v)$ não compartilham arestas, embora eles possam compartilhar nodos (veja a Figura 2.50). Portanto, a soma dos comprimentos dos caminhos associados aos nodos internos de T não é maior do que o número de arestas do *heap* T, ou seja, não mais do que $2n$. Concluímos que a construção *bottom-up* do *heap* leva tempo $O(n)$. ■

Resumindo, o Teorema 2.12 garante que o tempo de execução da primeira fase do *heapsort* pode ser reduzida até $O(n)$. Infelizmente, o tempo de execução da segunda fase do *heapsort* é $\Omega(n \log(n))$ no pior caso. Entretando, não justificaremos esse limite inferior até o Capítulo 4.

O padrão de projeto localizador

Concluímos esta seção discutindo um padrão de projeto que nos permite estender o TAD fila de prioridades para acrescentar funcionalidades que serão úteis, por exemplo, em alguns dos algoritmos de grafos discutidos neste livro.

Como vimos nas listas e árvores binárias, abstrair a informação posicional em um contêiner é uma ferramenta poderosa. O TAD posição, descrito na Seção 2.2.2, permite-nos identificar um "local" específico em um contêiner para armazenar um elemento. Uma posição pode ter seus elementos trocados, por exemplo, como conseqüência de uma operação swapElements, mas a informação de posição se mantém a mesma.

Também existem aplicações em que precisamos manter informações sobre os elementos à medida que eles são movidos em um contêiner. Um padrão de projeto que preenche essa necessidade é o ***localizador***. Um localizador é um mecanismo para manter a associação entre um elemento e sua posição corrente em um contêiner. Um localizador "gruda" em um determinado elemento, mesmo que esse elemento altere sua posição dentro do contêiner.

Um localizador é como um recibo de lavanderia: podemos entregar nossa roupa ao atendente e recebemos um recibo que é um localizador para nossa roupa. A localização de nossa roupa em relação a outras roupas pode mudar à medida que elas são entregues e retiradas, mas nosso recibo sempre pode ser usado para recuperar nossas roupas. O fato importante a lembrar sobre um localizador é que ele segue seu item associado, mesmo que ele troque de posição.

Assim como nosso recibo, podemos imaginar que recebemos algo de volta quando inserimos um elemento em um contêiner – podemos receber um localizador para o elemento. Esse localizador pode ser usado mais tarde para nos referirmos ao elemento dentro do contêiner, por exemplo, para especificar que esse elemento deve ser removido do contêiner. Como tipo abstrato de dados, um localizador ℓ suporta os seguintes métodos:

element(): retorna o elemento associado a ℓ.

key(): retorna a chave do item associado a ℓ.

Por uma questão de coerência, discutiremos agora como podemos usar localizadores para estender o repertório de operações do TAD fila de prioridades para incluir métodos que retornam localizadores e recebem localizadores como parâmetro.

Métodos para filas de prioridade com localizadores

Podemos usar localizadores de uma forma bastante natural no contexto de uma fila de prioridade. Um localizador, neste cenário, se mantém associado ao item inserido em uma fila de prioridade e permite-nos acessar o item de uma forma genérica independentemente da forma com a qual a fila de prioridade foi implementada. Essa capacidade é importante para a implementação de uma fila de prioridade, pois não existem posições intrínsecas nela, já que não nos referimos aos elementos através de "posições", "índices" ou "nodos".

Estendendo o TAD fila de prioridades

Usando localizadores, podemos estender o TAD fila de prioridades com os métodos a seguir para acessar e modificar uma fila de prioridades P:

min(): retorna o localizador do item de P que tem a menor chave.

insert(k,e): insere um item novo, com elemento e e chave k em P, e retorna um localizador para o item.

remove(ℓ): remove de P o item com localizador ℓ.

replaceElement(ℓ,e): substitui por e e retorna o item de P com localizador ℓ.

replaceKey(ℓ,k): susbtitui por k e retorna a chave do item de P com localizador ℓ.

Acessos baseados em localizadores executam em tempo $O(1)$, enquanto que um acesso baseado em chaves, que necessita procurar pelo elemento pesquisando em toda a seqüência ou *heap*, executa em tempo $O(n)$ no pior caso. Além disso, algumas aplicações restringem a operação replaceKey de forma que apenas incrementam ou decrementam chaves. Essa restrição pode ser feita pela definição de novos métodos increaseKey ou decreaseKey, por exemplo, que recebem um localizador por parâmetro. Outras aplicações de tais filas de prioridades são mostradas no Capítulo 7.

Comparando diferentes implementações de filas de prioridades

Na Tabela 2.51, comparamos os tempos de execução dos métodos do TAD fila de prioridades definidos nesta seção para seqüências não-ordenadas, seqüências ordenadas e implementações de *heap*.

Método	Seqüência não-ordenada	Seqüência ordenada	*Heap*
size, isEmpty, key, replaceElement	$O(1)$	$O(1)$	$O(1)$
minElement, min, minKey	$O(n)$	$O(1)$	$O(1)$
insertItem, insert	$O(1)$	$O(n)$	$O(\log n)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$

Tabela 2.51 Comparação dos tempos de execução dos métodos do TAD fila de prioridades para implementações de seqüências não-ordenadas, seqüências ordenadas e *heap*. Indicamos com n o número de elementos na fila de prioridade no momento em que o método é executado.

2.5 Dicionários e tabelas *hash*

Um dicionário de computador é semelhante a um dicionário de palavras na medida em que ambos são usados para localizar coisas. A idéia principal é que o usuário pode associar chaves a elementos, e então usar essas chaves mais tarde para localizar ou remover elementos (veja a Figura 2.52). Assim, o tipo abstrato de dados dicionário tem métodos para inserção, remoção e pesquisa de elementos usando chaves.

2.5.1 O TAD dicionário não-ordenado

Um dicionário armazena pares elemento-chave (k, e), que chamamos de ***itens***, onde k é a chave e e é o elemento. Por exemplo, em um dicionário que armazena registros de estudantes (tais como nome, endereço e notas), a chave pode ser o número de identificação do estudante. Em algumas aplicações, a chave pode ser o próprio elemento.

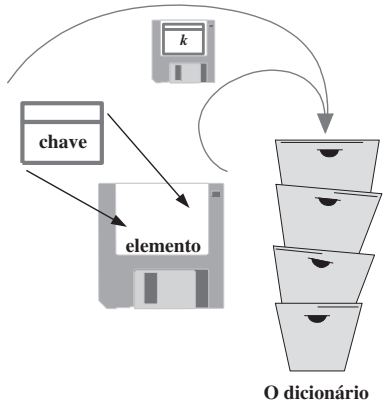


Figura 2.52 Uma figura conceitual do TAD dicionário. Chaves (rótulos) são atribuídas aos elementos (disquetes) pelo usuários. Os itens resultantes (disquetes rotulados) são inseridos no dicionário (arquivo de aço). As chaves podem ser usadas mais tarde para recuperar ou remover itens.

Distinguimos dois tipos de dicionários: o ***dicionário não-ordenado*** e o ***dicionário ordenado***. Estudaremos dicionários ordenados no Capítulo 3; discutiremos dicionários não-ordenados aqui. Em qualquer caso, consideramos a ***chave*** como um identificador que é atribuído por uma aplicação ou usuário a um elemento associado.

Por uma questão de generalidade, nossa definição permite que um dicionário armazene vários itens com a mesma chave. Entretanto, existem aplicações onde não queremos permitir itens com a mesma chave (por exemplo, em um dicionário que armazena os registros de estudantes, provavelmente não permitiremos que dois estudantes tenham o mesmo identificador). Nos casos em que as chaves são únicas, a chave associada a um objeto pode ser entendida como sendo o "endereço" daquele objeto na memória. Na verdade, tais dicionários são por vezes chamados de memória associativa, porque a chave associada a um objeto determina sua localização no dicionário.

Como um TAD, um ***dicionário*** *D* suporta os seguintes métodos:

findElement(k): se D contiver um item com a chave k, então o elemento deste item é retornado; senão, o elemento especial NO_SUCH_KEY é retornado.

insertItem(k,e): insere um item com elemento e e chave k em D.

removeElement(k): remove de D um item com chave igual a k, e retorna seu elemento. Se D não contiver um item com a chave k, então o elemento especial NO_SUCH_KEY é retornado.

Notamos que, se quisermos armazenar um item e em um dicionário de maneira que o item seja sua própria chave, então podemos inserir e usando a chamada de método insertItem(e,e). Quando as operações findElement(e) e removeElement(k) não são bem sucedidas (isto é, o dicionário D não tem itens com chave igual a k), usamos como convenção retornar o elemento especial NO_SUCH_KEY. Esse elemento especial é conhecido como ***sentinela***.

Além disso, um dicionário pode implementar outros métodos de suporte, tais como os métodos tradicionais de contêineres size() e isEmpty(). Além destes, podemos incluir o método elements(), que retorna os elementos armazenado em D e keys(), que retorna as chaves armazenadas em D. Assim, permitindo o uso de chaves não-exclusi-

vas, motivamos os métodos findAllElements(k), que retorna um iterador sobre todos os elementos com chave igual a k, e removeAllElements(k), que remove de D todos os itens com chave igual a k, a retornarem iteradores sobre seus elementos.

Arquivos de log

Uma forma simples de implementar um dicionário D usa uma seqüência não-ordenada S, que por sua vez é implementada usando um vetor ou uma lista para armazenar os pares elemento-chave. Esse tipo de implementação é normalmente chamada de ***arquivo de log***. As principais aplicações de um arquivo de log são aquelas situações em que desejamos armazenar pequenos conjuntos de dados ou dados que não mudam muito ao longo do tempo. Também podemos nos referir à implementação usando arquivo de log de D como ***implementação não-ordenada de seqüência***.

O espaço requerido por um arquivo de log é $\Theta(n)$, já que tanto um vetor quanto uma lista encadeada podem ter seu uso de memória proporcional ao número de elementos que eles armazenam. Adicionalmente, com uma implementação do TAD dicionário baseada em arquivo de log, podemos realizar a operação insertItem(k,e) fácil e eficientemente através de uma chamada ao método insertLast sobre S, que executa em tempo $O(1)$.

Infelizmente, essa implementação não permite uma execução eficiente do método findElement. Uma operação findElement(k) deve ser feita examinando-se toda a seqüência S, item por item. O pior caso para o tempo de execução deste método ocorre quando a pesquisa não é bem-sucedida, e alcançamos o fim da seqüência tendo examinado todos os n itens. Assim, o método findElement é executado em tempo $O(n)$. De forma similar, é necessário tempo linear no pior caso para a operação removeElement(k) em D, pois, para remover um item com chave k, temos primeiramente de encontrá-lo vasculhando a seqüência S.

2.5.2 Tabelas *hash*

As chaves associadas com elementos em um dicionário são freqüentemente consideradas como "endereços" dos elementos. Exemplos desse tipo de aplicação são as tabelas de símbolos de um compilador ou listas de variáveis de ambiente em um sistema operacional. Em ambos os casos, estas estruturas consistem em uma coleção de nomes simbólicos em que cada nome serve como "endereço" para propriedades sobre o tipo de uma variável ou seu valor. Uma das maneiras mais eficientes de implementar um dicionário em tais circunstâncias é usar uma ***tabela hash***. Embora, como veremos, o tempo de execução de pior caso das operações do TAD dicionário seja $O(n)$ quando usamos uma tabela *hash*, esta pode realizar tais operação em tempo esperado $O(1)$. Ela consiste em dois componentes principais, o primeiro deles sendo um arranjo de *buckets*.

Arranjos de *buckets*

Um ***arranjo de buckets*** para uma tabela *hash* é um arranjo A de tamanho N, no qual cada célula de A é considerada como um "*bucket*" (ou seja, um contêiner para pares chave-elemento), e o inteiro N determina a ***capacidade*** do arranjo. Se as chaves são inteiros bem distribuídos no intervalo $[0, N - 1]$, este arranjo de *buckets* é tudo o que é necessário. Um elemento e com chave k é simplesmente inserido no *bucket* $A[k]$.

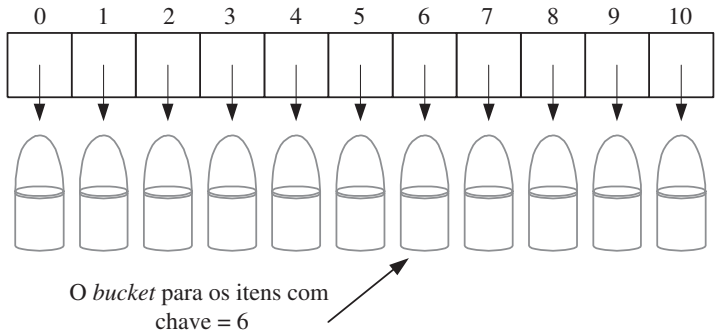


Figura 2.53 Uma ilustração do arranjo de *buckets*.

Quaisquer células associadas com chaves ausentes no dicionário são consideradas como contendo o objeto especial NO_SUCH_KEY (veja a Figura 2.53).

Naturalmente, se as chaves não são únicas, então dois elementos diferentes podem ser mapeados para o mesmo *bucket* em A. Neste caso, dizemos que ocorreu uma ***colisão***. É claro que, se cada *bucket* de A pode armazenar um único elemento, então não podemos associar mais de um elemento a cada *bucket*, o que é um problema no caso de colisões. Existem maneiras de tratar colisões, como veremos em seguida, mas a melhor estratégia é tentar evitá-las desde o início.

Análise do arranjo de *buckets*

Se as chaves são únicas, então as colisões não são um problema, e inserções, remoções e procuras na tabela *hash* levam um tempo $O(1)$ no pior caso. Este parece ser um grande resultado, mas tem duas desvantagens: a primeira é que ela usa espaço $Q(N)$, que não é necessariamente relacionado ao verdadeiro número n de itens presentes no dicionário. De fato, se N é grande em comparação a n, então esta implementação desperdiça espaço. A segunda desvantagem é que o arranjo de *buckets* exige que as chaves sejam diferentes inteiros no intervalo $[0, N - 1]$, o que freqüentemente não acontece. Já que essas duas desvantagens são tão comuns, definimos que a estrutura de dados tabela *hash* consiste em um arranjo de *buckets* e de um "bom" mapeamento de nossas chaves para inteiros no intervalo $[0, N - 1]$.

2.5.3 Funções *hash*

A segunda parte de uma tabela *hash* é uma função h, chamada de ***função hash***, que mapeia cada chave k em um inteiro no intervalo $[0, N - 1]$, onde N é a capacidade do arranjo de *buckets* para esta tabela. Com uma função *hash* h deste tipo, podemos aplicar o método do arranjo de *buckets* para chaves arbitrárias. A idéia central desta abordagem é usar o valor da função *hash* $h(k)$ como um índice no arranjo de *buckets* A, em vez da chave k (que é provavelmente inadequada para uso como índice de um arranjo de *buckets*). Ou seja, armazenamos o item (k, e) no *bucket* $A[h(k)]$.

Dizemos que uma função *hash* é "boa" se ela mapeia as chaves de nosso dicionário minimizando as colisões. Por razões práticas, também gostaríamos que o cálculo da função *hash* fosse fácil e rápido. Seguindo a convenção em Java, vemos a avaliação de $h(k)$ como consistindo em duas ações: mapear a chave k em um inteiro, chamado ***código hash***, e mapear o código *hash* em um inteiro dentro do intervalo de índices de um arranjo de *buckets*, chamado de ***mapa de compressão*** (veja a Figura 2.54).

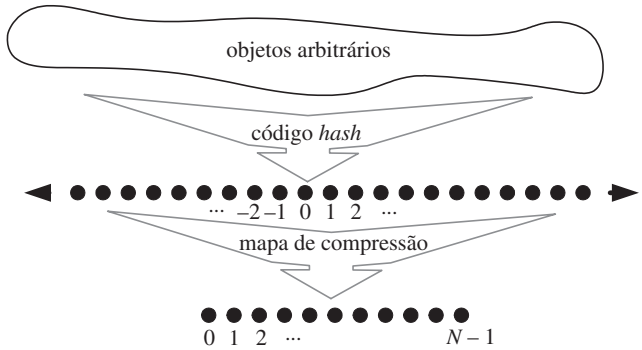


Figura 2.54 As duas partes de uma função *hash*: código *hash* e mapa de compressão.

Códigos *hash*

A primeira ação que uma função de *hash* realiza é tomar uma chave arbitrária k no dicionário e atribuir a ela um valor inteiro. O inteiro associado a uma chave k é chamado de ***código hash*** ou ***valor de hash*** para k. Esse inteiro não precisa estar no intervalo $[0, N-1]$ e pode mesmo ser negativo, mas desejamos que o conjunto de códigos *hash* associados a nossas chaves reduza as colisões tanto quanto possível. Além disso, para ser coerente com todas as nossas chaves, o código *hash* que usamos para uma chave k deve ser igual ao código *hash* de qualquer chave igual a k.

Soma dos componentes

Para tipos cuja representação em *bits* é duas vezes maior do que um código *hash*, a idéia acima não pode ser aplicada diretamente. Ainda assim, um código *hash* possível e usado por muitas implementações Java é simplesmente converter a representação de um inteiro longo para um inteiro do tamanho do código *hash*. Esse código *hash*, naturalmente, ignora metade da informação presente no valor original e, se muitas das chaves em nosso dicionário diferem apenas na outra metade, então elas colidirão através deste algoritmo simples. Um código *hash* alternativo que leva todos os *bits* em consideração é obtido somando-se a representação inteira dos *bits* de mais alta ordem e a representação inteira dos *bits* de mais baixa ordem. De fato, a alternativa baseada na soma de componentes pode ser estendida a qualquer objeto x cuja representação binária pode ser vista como uma k-tupla $(x_0, x_1,..., x_{k-1})$ de inteiros, pois podemos formar um código *hash* para x como $\sum_{i=0}^{k-1} x_i$.

Códigos *hash* polinomiais

O código *hash* baseado em somas descrito acima não é uma boa escolha para cadeias de caracteres ou outros objetos longos que podem ser vistos como tuplas da forma $(x_0, x_1,..., x_{k-1})$, onde a ordem dos elementos s_i é relevante. Por exemplo, considere um código *hash* para uma cadeia de caracteres s que soma os valores ASCII (ou Unicode) dos caracteres em s. Este código *hash* infelizmente produz muitas colisões indesejáveis para cadeias de caracteres bastante comuns. Em particular, "temp01" e "temp10" colidem com esta função e também com as palavras "stop", "pots", "spot" e "tops".

Um código *hash* melhor deveria levar em conta a posição dos elementos x_i. Uma alternativa que faz exatamente isso é escolher uma constante diferente de zero, $a \neq 1$ e usar como código *hash* o valor dado por

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1},$$

o qual, pela regra de Horner (veja o Exercício C-1.16), pode ser reescrito como

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots + a(x_2 + \mathrm{a}(x_1 + ax_0))\cdots)),$$

o qual, matematicamente falando, é simplesmente um polinômio em a que usa os componentes $(x_0, x_1,..., x_{k-1})$ como seus coeficientes. Este código *hash* é conhecido, portanto, como ***código hash polinomial***.

Análise experimental do código *hash*

Intuitivamente, um código *hash* polinomial usa a multiplicação pela constante a como uma forma de "dar espaço" a cada componente em uma tupla de valores, e ainda preserva a caracterização dos componentes anteriores. Em um computador típico, a avaliação de um polinômio será feita usando uma representação finita de *bits* para o código *hash*; assim, periodicamente o valor acumulado irá causar *overflow* nos *bits* usados para um inteiro. Entretanto, como estamos mais interessados na boa distribuição do objeto x em relação às outras chaves, simplesmente ignoramos este *overflow*. Ainda assim, devemos lembrar que este tipo de *overflow* é possível e escolher uma constante a que tenha alguns *bits* de baixa ordem diferentes de zero, o que servirá para preservar um pouco da informação mesmo em caso de *overflow*.

Fizemos alguns estudos experimentais que sugerem que 33, 37, 39 e 41 são valores particularmente bons para a quando as cadeias de caracteres a serem armazenadas são palavras da língua inglesa. De fato, em uma lista de mais de 50 mil palavras em inglês formada através da união de listas de palavras fornecidas em duas versões de Unix, constatamos que, escolhendo $a = 33$, 37, 39 ou 41, produz, tem-se menos de sete colisões em cada caso! Não deve ser uma surpresa, portanto, descobrir que várias versões de Java escolhem uma função *hash* polinomial baseada em uma dessas constantes. Para obter maior velocidade, no entanto, algumas implementações Java somente aplicam a função *hash* polinomial em uma fração de cadeias de caracteres muito longas, por exemplo, a cada oito caracteres.

2.5.4 Mapas de compressão

O código *hash* para uma chave k é tipicamente inadequado para uso imediato em um arranjo de *buckets*, porque o intervalo de códigos *hash* possíveis para as chaves será geralmente maior do que o intervalo de índices válidos no arranjo de *buckets* A. Ou seja, usar imediatamente o código *hash* como índice para o arranjo de *buckets* pode resultar no lançamento de uma exceção por acesso fora dos limites do arranjo se o índice for negativo ou se ultrapassar a capacidade de A. Assim, se determinarmos o código *hash* como um valor inteiro associado a uma chave k, ainda é preciso mapear esse inteiro para o intervalo $[0, N-1]$. Esta etapa de compressão é a segunda ação que uma função *hash* realiza.

O método da divisão

Um ***mapa de compressão*** bastante simples de usar é

$$h(k) = |k| \bmod N,$$

que é chamado de ***método da divisão***.

Se escolhermos N como sendo um número primo, então esta função *hash* ajuda a "espalhar" a distribuição dos valores. De fato, se N não for primo, então existe uma maior probabilidade de que padrões na distribuição das chaves sejam repetidos na distribuição dos códigos *hash*, causando colisões. Por exemplo, se tivermos as chaves {200, 205, 210, 215, 220,..., 600} em um arranjo de *buckets* de tamanho 100, então cada código *hash* irá colidir com três outros. Se este mesmo conjunto de chaves for colocado em um arranjo de *buckets* de tamanho 101, no entanto, não haverá colisões. Se uma função *hash* é bem escolhida, ela deveria garantir que a probabilidade de duas chaves diferentes irem para a mesma posição no arranjo de *buckets* é de no máximo $1/N$. Escolher N como um número primo não é sempre suficiente, no entanto, pois, se há um padrão repetitivo de chaves com formato $iN + j$ para vários valores diferentes de i, então ainda teremos colisões.

O método MAD

Uma função de compressão mais sofisticada que ajuda a eliminar padrões repetitivos em um conjunto de chaves inteiras é o método de ***multiplicação, adição e divisão*** (ou "MAD"). Usando este método, definimos a função de compressão como

$$h(k) = |ak + b| \bmod N,$$

onde N é um número primo, e a e b são inteiros não-negativos escolhidos aleatoriamente quando a função de compressão é determinada, de forma que $a \bmod N \neq 0$. Esta função de compressão é escolhida de forma a eliminar padrões repetidos no conjunto de códigos *hash* e levar-nos mais perto de uma "boa" função *hash*, ou seja, uma função em que a probabilidade de colisão de duas chaves é de no máximo $1/N$. Este comportamento seria o mesmo que teríamos se as chaves fossem "jogadas" em A de forma aleatória e uniforme.

Com uma função de compressão como esta, que espalha n inteiros de forma bastante homogênea no intervalo $[0, N - 1]$, e um mapeamento das chaves em nosso dicionário para os números inteiros, temos uma função *hash* efetiva. Juntos, uma função como essa e um arranjo de *buckets* definem os componentes essenciais de uma implementação baseada em tabela *hash* para o TAD dicionário.

Antes que entremos nos detalhes de como realizar operações como findElement, insertItem e removeElement, devemos primeiramente resolver o problema do tratamento de colisões.

2.5.5 Métodos para tratamento de colisões

Lembramos que a idéia principal de uma tabela *hash* é usar um arranjo de *buckets* A e uma função *hash* h e usá-los para implementar um dicionário armazenando cada item (k,e) na posição $A[h(k)]$. Esta idéia simples torna-se complicada, no entanto, quando temos duas chaves distintas k_1 e k_2 com $h(k_1) = h(k_2)$. A existência desta ***colisão*** impede que façamos imediatamente a inserção do novo item (k,e) na posição $A[h(k)]$. Ela tam-

bém complica a operação findElement(k). Para lidar com colisões e desenvolver formas corretas de implementar os métodos do TAD dicionário, devemos ter uma estratégia consistente para tratar colisões.

Encadeamento separado

Uma maneira simples e eficiente de lidar com colisões é fazer com que cada posição $A[i]$ armazene uma referência para uma lista, vetor ou seqüência S_i que armazena todos os elementos que nossa função *hash* mapeou para a posição $A[i]$. A seqüência S_i pode ser vista como um dicionário em miniatura, implementada usando-se uma seqüência não-ordenada ou arquivo de *log*, mas restrita a armazenar itens (k,e) tais que $h(k) = i$. Esta regra para ***resolução de colisões*** é conhecida como ***encadeamento separado***. Assumindo que implementamos cada posição não-vazia do arranjo de *buckets* A como um dicionário miniatura usando arquivos de *log*, podemos realizar as operações essenciais para um dicionário como segue:

- findElement(k):
  $B \leftarrow A[h(k)]$
  se B estiver vazio **então**
    retorne NO_SUCH_KEY
  senão
    {procura pela chave k na seqüência deste *bucket*}
    retorne B.findElement(k)

- insertItem(k,e):
  se $A[h(k)]$ estiver vazio **então**
    Cria um novo dicionário B, inicialmente vazio, baseado em seqüência
    $A[h(k)] \leftarrow B$
  senão
    $B \leftarrow A[h(k)]$
  B.insertItem(k,e)

- removeElement(k):
  $B \leftarrow A[h(k)]$
  se B estiver vazio **então**
    retorne NO_SUCH_KEY
  senão
    retorne B.removeElement(k)

Assim, para cada uma das operações fundamentais de dicionários envolvendo uma chave k, delegamos o tratamento desta operação ao dicionário miniatura baseado em seqüência armazenado em $A[h(k)]$. Assim, uma inserção colocará o novo item ao final da seqüência, uma procura examinará a seqüência até chegar a seu final ou encontrar o item procurado e uma remoção retirará o item depois que ele for encontrado. Podemos usar um dicionário baseado em arquivo de *log* desta maneira porque as propriedades de distribuição das chaves na função *hash* ajudam a manter pequeno cada dicionário miniatura. De fato, uma boa função *hash* tenta minimizar colisões tanto quanto possível, o que implica que a maior parte das posições do arranjo de *buckets* estarão vazias ou contendo apenas um elemento.

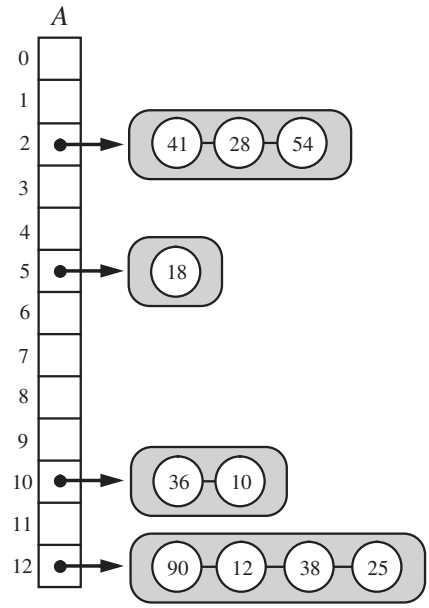


Figura 2.55 Ilustramos uma tabela *hash* de tamanho 13, armazenando 10 chaves inteiras, com colisões, resolvidas pelo método de encadeamento. O mapa de compressão, neste caso, é $h(k) = k \bmod 13$.

Na Figura 2.55, apresentamos uma tabela *hash* simples que usa uma função de compressão por divisão e encadeamento separado para resolver colisões.

Fatores de carga e *rehashing*

Supondo que estamos usando uma boa função *hash* para colocar os n itens de nosso dicionário em um arranjo de *buckets* de tamanho N, esperamos que o número de elementos associados a cada posição de A seja $[n/N]$. Esse parâmetro, que é chamado de ***fator de carga*** da tabela *hash*, deveria, portanto, ser limitado por uma constante pequena, preferencialmente 1. Assim, dada uma boa função *hash*, o tempo de execução esperado das operações findElement, insertItem e removeElement em um dicionário implementado com uma tabela *hash* que usa esta função é $O([n/N])$. Assim, podemos implementar operações padrão de um dicionário que são executadas em tempo $O(1)$, se soubermos que n é $O(N)$.

Manter o fator de carga de uma tabela *hash* constante (0,75 é normal) requer trabalho extra sempre que adicionamos elementos que excedam este limite. Nestes casos, para manter o fator de carga abaixo da constante especificada, necessitamos aumentar o tamanho do arranjo de *bucket* e trocar o mapa de compressão para suportar este novo tamanho. Além disso, devemos inserir todos os elementos da tabela *hash* existente no novo arranjo do *bucket* usando o novo mapa de compressão. Essa alteração de tamanho e reconstrução da tabela é chamada de ***rehashing***. Seguindo a abordagem do arranjo extensível (Seção 1.5.2), uma boa escolha é executar o *rehashing* em um arranjo com aproximadamente o dobro do tamanho do arranjo original, escolhendo um número primo como tamanho do novo arranjo.

Endereçamento aberto

O método do encadeamento separado tem muitas propriedades úteis, tais como permitir implementações simples das operações de dicionários, mas mesmo assim, ela tem uma pequena desvantagem: ela requer o uso de uma estrutura de dados auxiliar — uma lista, vetor ou seqüência — para armazenar itens em um arquivo de *log* para casos de colisão. No entanto, podemos tratar as colisões de outras maneiras. Em particular, se o espaço for precioso (por exemplo, se estamos escrevendo *software* para um dispositivo de pequeno porte), então podemos usar uma abordagem alternativa e sempre armazenar cada item diretamente no arranjo de *buckets*, com um item em cada posição. Essa abordagem economiza espaço porque estruturas de dados auxiliares não são necessárias, mas requer maior complexidade no tratamento de colisões. Existem vários métodos para implementar esta abordagem, que são chamados de métodos de ***endereçamento aberto***.

Teste linear

Uma estratégia simples para o tratamento de colisões com endereçamento aberto é o ***teste linear***. Nesta estratégia, se tentarmos inserir um item (k,e) em uma posição $A[i]$ que já estiver ocupada, onde $i = h(k)$, então tentamos de novo em $A[(i + 1) \bmod N]$. Se $A[(i + 1) \bmod N]$ estiver ocupada, então tentamos $A[(i + 2) \bmod N]$, e assim por diante, até acharmos uma posição que possa aceitar um novo item. Quando esta posição é localizada simplesmente inserimos o item (k,e) nela. É claro que, se usarmos esta estratégia de inserção, temos de alterar a implementação da operação findElement. Em particular, para realizar esta operação, temos de examinar posições consecutivas, começando em $A[h(k)]$, até achar um item com chave igual a k ou uma posição vazia (e neste caso, a procura é encerrada) (veja a Figura 2.56).

A operação removeElement(k) é mais complicada, no entanto. De fato, para implementar completamente este método, devemos restaurar o conteúdo da posição para parecer que o item com chave k nunca foi inserido. Embora fazer esta restauração seja possível, ela requer que movamos os elementos após $A[i]$, sem poder mover quaisquer outros que já estejam em suas posições corretas. Uma maneira típica de contornar esta dificuldade é substituir o item deletado por um objeto representando um "item desativado". Esse objeto deve ser marcado de alguma forma para que possamos detectar quando ele está ocupando uma posição. Com este marcador especial ocupando posições em nossa tabela *hash*, modificamos nosso algoritmo de procura em removeElement(k) e findElement(k) de forma que a procura pela chave k ignore itens desativados e continue a procura até achar o item ou uma posição vazia. Por sua vez, o algoritmo insertItem(k,e) deve parar em uma posição desativada e colocar nela o novo item.

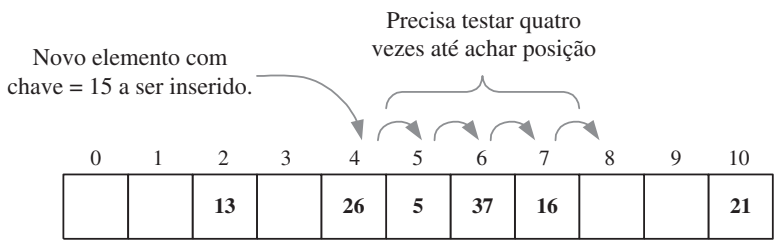


Figura 2.56 Inserção em uma tabela *hash* usando teste linear para resolver colisões. Aqui usamos o mapa de compressão $h(k) = k \bmod 11$.

Teste quadrático

Outra estratégia de endereçamento aberto é conhecida como ***teste quadrático*** e envolve o teste das posições $A[(i + f(j)) \bmod N]$, para $j = 0,1,...$, onde $f(j) = j^2$, até que seja achada uma posição vazia. Como no teste linear, o teste quadrático complica a operação de remoção, mas evita os problemas de agrupamento que acontecem com o teste linear. Mesmo assim, ela cria seu próprio padrão de agrupamento, chamado de ***agrupamento secundário***, em que o conjunto de posições ocupadas no arranjo de *buckets* "salta" no arranjo de maneira predeterminada. Se N não for um número primo, então o teste quadrático pode falhar em achar uma posição, mesmo que existam posições livres no arranjo. De fato, mesmo se N for primo, esta estratégia pode não achar uma posição livre se o arranjo de *buckets* estiver com mais de 50% de ocupação.

Hashing duplo

Outra estratégia de endereçamento aberto que não causa agrupamento do tipo produzido pelo teste linear ou pelo teste quadrático é o ***hashing duplo***. Nesta abordagem, escolhemos uma função *hash* secundária h', e se h leva alguma chave k a uma posição $A[i]$ (com $i = h(k)$) que já estiver ocupada, então tentamos as posições $A[(i+f(j)) \bmod N]$, para $j = 1,2,...$, onde $f(j) = j \cdot h'(k)$. Neste esquema, a função de *hash* secundária não pode resultar em zero, e uma escolha comum é $h'(k) = q - (k \bmod q)$ para algum número primo $q < N$. N também deve ser primo. Além disso, devemos escolher uma função *hash* secundária que minimize os agrupamentos tanto quanto possível.

Essas técnicas de ***endereçamento aberto*** economizam algum espaço se comparadas com a técnica de encadeamento separado, mas não são necessariamente mais rápidas. Em análises teóricas e experimentais, o método de encadeamento é competitivo ou mais rápido do que os outros métodos, dependendo do fator de carga do arranjo de *buckets*. Assim, se o uso de memória não for um problema, então o método de tratamento de colisões a escolher parece ser o encadeamento aberto. Mesmo assim, se houver pouca memória, então um destes métodos mais lentos pode ser implementado, desde que nossa estratégia de teste minimize a possibilidade de formação de agrupamentos decorrente do endereçamento aberto.

2.5.6 *Hashing* universal

Nesta seção, mostraremos como garantir que uma função *hash* é boa. Para fazê-lo com o devido cuidado, entretanto, teremos de tornar nossa discussão um pouco mais formal.

Como já foi dito, podemos assumir, sem perda de generalidade, que nosso conjunto de chaves são inteiros pertencentes a algum intervalo. Seja $[0, M - 1]$ este intervalo. Assim podemos ver uma função *hash* h como um mapeamento de inteiros pertencentes ao intervalo $[0, M - 1]$ para inteiros pertencentes ao intervalo $[0, N - 1]$ e consideraremos o conjunto de funções *hash* candidatas como sendo a ***família*** H de funções *hash*. Esta família é ***universal*** para quaisquer dois inteiros j e k pertencentes a $[0, M - 1]$ e para funções *hash* escolhidas de forma aleatóriamente uniforme de H,

$$\Pr(h(j) = h(k)) \leq \frac{1}{N}$$

Essa família também é conhecida como família ***2-universal*** de funções *hash*. O propósito de se escolher uma boa função *hash* pode, entretanto, ser visto como o problema de selecionar uma pequena família H universal de funções *hash* que são fáceis de calcular. A razão pela qual famílias de funções *hash* são úteis é que elas possuem uma baixa expectativa para o número de colisões.

Teorema 2.13: *seja j um inteiro no intervalo* $[0, M-1]$, *seja S um conjunto de n inteiros neste mesmo intervalo e seja h uma função hash escolhida de forma uniformemente aleatória a partir de uma família de funções hash universais a partir de inteiros no intervalo* $[0, M-1]$ *até inteiros no intevalo* $[0, N-1]$. *Desta forma, o número de colisões entre j e os inteiros em S é no máximo n/N.*

Prova: faça $c_h(j,S)$ indicar o número de colisões entre j e inteiros em S (ou seja, $c_h(j, S) = |\{k \in S : h(j) = h(k)\}|$). A quantidade em que estamos interessados é o valor esperado $E(c_h(j,S))$. Podemos escrever $c_h(j,S)$ como

$$c_h(j,S) = \sum_{k \in S} X_{j,k} \ ,$$

onde $X_{j,k}$ é uma variável aleatória que assume valor 1 se $h(j) = h(k)$ e valor 0 em qualquer outro caso (isto é, $X_{j,k}$ é uma variável aleatória ***indicadora*** de colisões entre j e k). Pela linearidade da expectativa,

$$E(c_h(j,S)) = \sum_{s \in S} E(X_{j,k})$$

Também, pela definição de uma família universal, $E(X_{j,k}) \leq 1/N$. Assim,

$$E(c_h(j,S)) \leq \sum_{s \in S} \frac{1}{N} = \frac{n}{N}$$

■

Dito de outra forma, esse teorema afirma que o número esperado de colisões entre o código *hash j* e as chaves que já estão na tabela *hash* (usando uma função *hash* escolhida aleatoriamente a partir de uma família universal H) é, no máximo, o fator corrente de carga da tabela *hash*. Uma vez que o tempo para executar uma pesquisa, inserção ou deleção para uma chave j em uma tabela *hash* que usa o encadeamento da regra de resolução de colisões é proporcional à quantidade de colisões entre j e outras chaves na tabela, isso implica que o tempo esperado de execução de qualquer uma das operações é proporcional ao fator de carga da tabela *hash*. Isso é exatamente o que queremos.

Vamos focar nossa atenção, então, no problema de construir uma pequena família de funções *hash* universais que sejam fáceis de calcular. O conjunto de funções *hash* que construímos é, na verdade, parecido a família final que consideramos no final da seção anterior. Seja p um número primo maior ou igual ao número de códigos

hash M mas menor que $2M$ (e sempre existe este número de acordo com o ***Postulado de Bertrand***).

Defina H como o conjunto de funções *hash* da forma

$$h_{a,b}(k) = (ak + b \bmod p) \bmod N.$$

O teorema a seguir estabelece que esta família de funções *hash* é universal.

Teorema 2.14: *a família* $H = \{h_{a,b}: 0 < a < p \text{ e } 0 \leq b < p\}$ *é universal.*

Prova: seja Z o conjunto de inteiros pertencentes ao intervalo $[0, p - 1]$. Vamos separar cada função *hash* $h_{a,b}$ nas funções

$$f_{a,b}(k) = ak + b \bmod p$$

e

$$g(k) = k \bmod N,$$

de forma que $h_{a,b} = g(f_{a,b}(k))$. O conjunto de funções $f_{a,b}$ define uma família de funções *hash* F que mapeiam inteiros em Z para inteiros em Z. Afirmamos que cada função em F não causa colisões. Para justificar essa afirmativa, considere $f_{a,b}(j)$ e $f_{a,b}(k)$ para algum par de inteiros diferentes j e k pertencentes a Z. Se $f_{a,b}(j) = f_{a,b}(k)$, então poderemos ter uma colisão. Mas, relembrando a definição do módulo da operação isso, irá implicar que

$$aj + b - \left\lfloor \frac{aj + b}{p} \right\rfloor p = ak + b - \left\lfloor \frac{ak + b}{p} \right\rfloor p.$$

Sem perder generalidade, podemos assumir que $k < j$, o que implica que

$$a(j - k) = \left(\left\lfloor \frac{aj + b}{p} \right\rfloor - \left\lfloor \frac{ak + b}{p} \right\rfloor \right) p$$

Uma vez que $a \neq 0$ e $k < j$, isso, por sua vez, implica que $a(j - k)$ é múltiplo de p. Mas $a < p$ e $j - k$, então não existe maneira pela qual $a(j - k)$ possa ser um número positivo múltiplo de p, porque p é primo (lembre-se de que todo inteiro positivo pode ser fatorado em um produto de primos). Assim, é impossível que $f_{a,b}(j) = f_{a,b}(k)$ se $j \neq k$. Colocando isso de outra forma, cada $f_{a,b}$ mapeia inteiros de Z para inteiros em Z de forma a determinar uma correspondência de um para um. Desde que as funções em F não causem colisões, a única forma que uma função $h_{a,b}$ pode causar uma colisão é se a função g o fizer.

Sejam j e k dois inteiros diferentes pertencentes a Z. Seja $c(j,k)$ o número de funções em H que mapeiam j e k para o mesmo inteiro (isto é, que fazem j e k colidirem). Podemos derivar o limite superior de $c(j,k)$ usando um parâmetro contador simples. Se consideramos qualquer inteiro x em Z, existem p funções $f_{a,b}$ diferentes tais que $f_{a,b} = x$ (uma vez que podemos escolher um b para cada a). Fixando x, notamos que cada função $f_{a,b}$ mapeia k para um único inteiro

$$y = f_{a,b}(k)$$

em Z com $x \neq y$. Além disso, quanto aos p inteiros distintos da forma $y = f_{a,b}(k)$, existem no máximo

$$\lceil p / N \rceil - 1$$

de forma que $g(y) = g(x)$ e $x \neq y$ (pela definição de g). Assim, para qualquer x pertencente a Z, existem no máximo $\lceil p/N \rceil - 1$ funções $h_{a,b}$ em H de forma que

$$x = f_{a,b}(j) \quad \text{e} \quad h_{a,b}(j) = h_{a,b}(k).$$

Uma vez que existem p escolhas para o inteiro x em Z, a contagem de argumentos acima implica que

$$\begin{aligned} c(j,k) &\leq p\left(\left\lceil \frac{p}{N} \right\rceil - 1\right) \\ &\leq \frac{p(p-1)}{N}. \end{aligned}$$

Existem $p(p-1)$ funções em H, uma vez que cada função $h_{a,b}$ é determinada por um par (a,b) de forma que $0 < a < p$ e $0 \leq b < p$. Assim, selecionar uma função aleatoriamente de H implica selecionar uma das $p(p-1)$ funções. Por isso, para qualquer dois inteiros diferentes j e k em Z,

$$\begin{aligned} \Pr(h_{a,b}(j) = h_{a,b}(k)) &\leq \frac{p(p-1)/N}{p(p-1)} \\ &= \frac{1}{N}. \end{aligned}$$

Isto é, a família H é universal. ■

Além de serem universais, as funções em H possuem algumas outras propriedades interessantes. Todas funções em H são fáceis de selecionar, uma vez que, para tanto, requerem simplesmente que selecionemos um par de inteiros aleatórios a e b de forma que $0 < a < p$ e $0 \leq b < p$. Além disso, cada função em H é fácil de calcular em tempo $O(1)$, requerendo apenas uma multiplicação, uma adição e duas aplicações da função módulo. Assim, qualquer função *hash* escolhida aleatoriamente em H irá resultar em uma implementação do TAD dicionário, onde todas as operações fundamentais têm tempo esperado de execução $O(\lceil n/N \rceil)$, uma vez que estamos usando a regra de encadeamento para a resolução de colisões.

2.6 Exemplo em Java: *heap*

Para ilustrar melhor como os métodos do TAD árvore e do TAD fila de prioridades podem interagir em uma implementação concreta da estrutura de dados *heap*, discutiremos, nesta seção, a implementação do estudo de caso de uma estrutura de dados *heap* escrita em Java. Uma implementação Java de uma fila de prioridades que usa um *heap* é apresentada nos Algoritmos 2.57 a 2.60. Para auxiliar na modularização, delegamos a manutenção da estrutura do *heap* propriamente dito para a estrutura de dados chamada ***árvore-heap***, que estende uma árvore binária e provê os seguintes métodos de atualização adicionais:

add(o): executa a seguinte seqüência de operações:
expandExternal(z);
replaceElement(z,o);
return z;
de forma que z se transforme no último nodo da árvore ao final da operação.

remove(): executa a seguinte seqüência de operações:
$t \leftarrow z$.element();
removeAboveExternal(rigthChild(z));
return t;
onde z é o último nodo no início da operação.

Ou seja, a operação add adiciona um elemento novo no primeiro nodo externo, enquanto que a operação remove remove o elemento do último nodo. Usando uma implementação de árvore baseada em vetor (veja Seção 2.3.4), as operações add e remove consomem tempo $O(1)$. O TAD árvore-*heap* é representado pela interface Java HeapTree apresentada no Algorimto 2.57. Assumimos que a classe Java VectorHeapTree (que não é mostrada) implementa a interface HeapTree usando um vetor e suportando os métodos add e remove em tempo $O(1)$.

```
public interface HeapTree extends InspectableBinaryTree, PositionalContainer {
    public Position add(Object elem);
    public Object remove();
}
```

Trecho de Código 2.57 Interface HeapTree para uma árvore-*heap*. Estende a interface InspectableBinaryTree com os métodos replaceElement e swapElements, herdados da interface PositionalContainer, e acrescenta os métodos especializados de atualização add e remove.

A classe HeapPriorityQueue implementa a interface PriorityQueue usando um *heap*. Ela é mostrada nos Algoritmos 2.58 e 2.60. Observe que armazenamos itens elemento-chave da classe Item, que é simplesmente uma classe para pares elemento-chave em uma árvore-*heap*.

```
public class HeapPriorityQueue implements PriorityQueue {
  HeapTree T;
  Comparator comp;

  public HeapPriorityQueue(Comparator c) {
    if ((comp = c) == null)
      throw new IllegalArgumentException("Null comparator passed");
    T = new VectorHeapTree();
  }

  public int size() {
    return (T.size() – 1) / 2;
  }
```

```
  public boolean isEmpty() {
    return T.size() == 1; }

  public Object minElement() throws PriorityQueueEmptyException {
    if (isEmpty())
      throw new PriorityQueueEmptyException("Empty Priority Queue");
    return element(T. root());
  }

  public Object minKey() throws PriorityQueueEmptyException {
    if (isEmpty())
      throw new PriorityQueueEmptyException("Empty Priority Queue");
    return key(T.root());
  }

⋮
```

Trecho de Código 2.58 Variáveis de instância, construtor e métodos size, isEmpty, minElement e minKey da classe HeapPriorityQueue, que implementa uma fila de prioridades usando um *heap*. Outros métodos desta classe são mostrados no Algoritmo 2.60. Os métodos auxiliares key e element extraem a chave e o elemento de um item da fila de prioridades armazenado em uma determinada posição da árvore *heap*.

```
public void insertItem (Object k, Object e) throws InvalidKeyException {
  if (!comp.isComparable(k))
    throw new InvalidKeyException("Invalid Key");
  Position z = T.add(new Item(k, e));
  Position u;
  while (!T.isRoot(z)) { // up-heap bubbling
    u = T.parent(z);
    if (comp.isLessThanOrEqualTo(key(u), key(z)))
      break;
    T.swapElements(u, z);
    z = u;
  }
}

public Object removeMin() throws PriorityQueueEmptyException {
  if (isEmpty())
    throw new PriorityQueueEmptyException("Empty priority queue!");
  Object min = element(T. root());
  if (size() == 1)
    T.remove();
  else {
    T. replaceElement(T. root(), T.remove());
    Position r = T.root();
    while (T.isInternal(T.leftChild(r))) {  // down-heap bubbling
      Position s;
      if (T.isExternal(T.rightChild(r)) ||
          comp.isLessThanOrEqualTo(key(T.leftChild(r)), key(T.rightChild(r))))
        s = T.leftChild(r);
```

```
        else
          s = T.rightChild(r);
        if (comp.isLessThan(key(s), key(r))) {
          T.swapElements(r, s);
          r = s;
        }
        else
          break;
      }
    }
    return min;
}
```

Trecho de Código 2.60 Métodos insertItem e removeMin da classe HeapPriorityQueue. Outros métodos desta classe são apresentados no Algoritmo 2.58.

2.7 Exercícios

Reforço

R-2.1 Descreva, usando pseudocódigo, implementações para os métodos insertBefore(p,e), insertFirst(e) e insertLast(e) do TAD lista, considerando que a lista é implementada usando uma lista duplamente encadeada.

R-2.2 Desenhe uma árvore de expressão que tenha quatro nodos externos armazenando os números 1, 5, 6 e 7 (com um número armazenado em cada nodo externo, mas não necessariamente nesta ordem), e com três nodos internos, cada um armazenando uma operação pertencente ao conjunto $\{+, -, \times, /\}$ de operações aritméticas binárias, de maneira que o valor desta árvore seja 21. Presume-se que os operadores retornam números racionais (não-inteiros) e que um operador pode ser usado mais de uma vez (mas apenas um operador é armazenado em cada nodo interno).

R-2.3 Seja T uma árvore ordenada com mais de um nodo. É possível que o caminhamento prefixado de T visite os nodos na mesma ordem que o caminhamento pós-fixado de T? Em caso positivo, apresente um exemplo; em caso negativo, argumente por que isso não pode ocorrer. Da mesma forma, é possível que o caminhamento prefixado de T visite os nodos na ordem inversa do caminhamento pós-fixado de T? Em caso positivo, apresente um exemplo; em caso negativo, argumente por que isso não pode ocorrer.

R-2.4 Responda às questões a seguir de maneira a justificar o Teorema 2.8.

a. Desenhe uma árvore binária de altura 7 com o máximo de nodos externos.
b. Qual é o número mínimo de nodos externos para uma árvore binária de altura h? Justifique.
c. Qual é o número máximo de nodos externos para uma árvore binária de altura h? Justifique.

d. Seja T uma árvore binária com altura h e n nodos. Mostre que

$$\log(n + 1) - 1 \leq h \leq (n - 1)/2.$$

e. Para quais valores de n e h os limites superior e inferior de h podem ser igualmente alcançados?

R-2.5 Seja T uma árvore binária tal que todos os nodos externos tenham a mesma altura. Seja D_e a soma das profundidades de todos os nodos externos de T e seja D_i a soma das profundidades de todos os nodos internos de T. Determine as constantes a e b que satisfaçam

$$D_e + 1 = aD_i + bn,$$

onde n é o número de nodos de T.

R-2.6 Seja T uma árvore binária com n nodos e seja p a numeração dos níveis dos nodos de T como apresentado na Seção 2.3.4.

a. Mostre que, para cada nodo v de T, $p(v) \leq 2^{(n+1)/2} - 1$.
b. Apresente um exemplo de árvore binária com no mínimo cinco nodos que alcancem o limite superior no valor máximo de $p(v)$ para algum nodo v.

R-2.7 Seja T uma árvore binária com n nodos implementada usando um vetor S e seja p a numeração dos níveis dos nodos de T, como apresentado na Seção 2.3.4. Forneça o pseudocódigo para cada um dos métodos root, parent, leftChild, rightChild, isInternal, isExternal e isRoot.

R-2.8 Ilustre o desempenho do algoritmo de seleção ordenada sobre a seqüência de entrada: (22, 15, 36, 44, 10, 3, 9, 13, 29, 25).

R-2.9 Ilustre o desempenho do algoritmo de inserção ordenada sobre a seqüência de entrada do problema anterior.

R-2.10 Apresente um exemplo de uma seqüência com n elementos que corresponda ao pior caso para o algoritmo de inserção ordenada e demonstre que a inserção ordenada executa em tempo $\Omega(n^2)$ sobre esta seqüência.

R-2.11 De que maneira o item com a maior chave pode ser armazenado em um *heap*?

R-2.12 Demonstre a performance do algoritmo de *heap-sort* na seguinte seqüência de entrada: (2, 5,16, 4,10, 23, 39,18, 26,15).

R-2.13 Suponha que uma árvore binária T é implementada usando um vetor S, como descrito na Seção 2.3.4. Se n itens são armazenados em S de forma ordenada, iniciando pelo índice 1, esta árvore corresponde a um *heap*?

R-2.14 Existe um *heap* T que armazena sete elementos distintos de maneira que o caminhamento prefixado de T retorne os elementos de T ordenados? E no caso de um caminhamento interfixado? E no de um pós-fixado?

R-2.15 Mostre que o somatório $\Sigma_{i=1}^{n} \log i$ que aparece na análise do *heap-sort* é $\Omega(n \log n)$.

R-2.16 Mostre os passos para a remoção da chave 16 do *heap* da Figura 2.41.

R-2.17 Mostre os passos para substituir 5 por 18 no *heap* da Figura 2.41.

R-2.18 Desenhe um exemplo de *heap* cujas chaves são todas números ímpares entre 1 e 59 (sem repetições), de forma que a inserção do item com chave 32 causaria uma reordenação para cima de forma a percorrer o caminho do filho até a raiz (substituindo a chave do filho por 32).

R-2.19 Desenhe o item 11 de uma tabela *hash* resultante do *hashing* das chaves 12, 44, 13, 88, 23, 94, 11,39,20, 16 e 5 usando a função *hash* $h(i) = (2i + 5) \bmod 11$ supondo que as colisões são tratadas por encadeamento.

R-2.20 Qual seria o resultado do exercício anterior se as colisões fossem tratadas por teste linear?

R-2.21 Mostre o resultado do Exercício 2.19 supondo que as colisões são tratadas por teste quadrático, até o ponto em que o método falha porque nenhum local vazio é encontrado.

R-2.22 Qual o resultado do Exercício 2.19 supondo que as colisões são tratadas por *hashing* duplo usando como função secundária de *hash* a função $h'(k) = 7 - (k \bmod 7)$?

R-2.23 Apresente a descrição em pseudocódigo de uma inserção em uma tabela *hash* que usa teste quadrático para resolver colisões, supondo que também usamos o truque de substituir os itens deletados por um objeto especial "desativado".

R-2.24 Apresente o resultado do *rehashing* da tabela *hash* apresentada na Figura 2.55 em uma tabela de tamanho 19 usando uma nova função *hash* $h(k) = 2k \bmod 19$.

Criatividade

C-2.1 Descreva, usando pseudocódigo, um método de busca pelos *links* para determinar o nodo do meio de uma lista duplamente encadeada que use sentinelas no início e no fim, com número ímpar de nodos reais entre elas. (Observação: este método só deve usar os *links*; não é permitido usar contadores.) Qual o tempo de execução deste método?

C-2.2 Descreva como implementar o TAD fila usando duas pilhas, de forma que o tempo de execução amortizado para queue e dequeue seja $O(1)$, supondo que as pilhas suportem tempo constante para as operações push, pop e size. Qual o tempo de execução para os métodos queue e dequeue neste caso?

C-2.3 Demonstre como implementar o TAD pilha usando duas pilhas. Qual o tempo de execução dos métodos push, pop e size neste caso?

C-2.4 Descreva um algoritmo recursivo que enumere todas as permutações dos números $\{1, 2, ..., n\}$. Qual o tempo de execução deste método?

C-2.5 Descreva a estrutura e o pseudocódigo para uma implementação que usa arranjos do TAD vetor que obtenha tempo $O(1)$ para inserções e remoções na colocação 0, bem como nas inserções e remoções no fim do vetor. Sua implementação deve prover, também, um método elemAtRank de tempo constante.

C-2.6 No jogo infantil "batata quente" um grupo de n crianças senta em círculo passando um objeto chamado de "batata", ao redor do círculo (digamos no sentido horário). As crianças passam a batata até que o líder toque um sino, o que indica que a criança que estiver segurando a batata deve deixar o jogo, e as outras crianças reduzem o círculo. Esse processo continua até que só reste uma criança, que é declarada vencedora. Usando o TAD seqüência, descreva um método eficiente para implementar este jogo. Suponha que o líder sempre toque a campainha após a batata ter sido passada k vezes. (A determinação de qual será a criança que sobra nesta variação do jogo da "batata quente" é conhecido como o ***Problema de Josephus***.) Qual o tempo de execução de seu método em termos de n e m, supondo que a seqüência é implementada usando uma lista duplamente encadeada? E se a seqüência for implementada usando um arranjo?

C-2.7 Usando o TAD Sequence, descreva uma forma eficiente de representar um conjunto de n cartas embaralhadas usando uma seqüência. Use a função randomInt(n), que retorna um número randômico entre 0 e $n - 1$, inclusive. Seu método deve garantir que todas as possíveis ordenações tenham igual probabilidade. Qual o tempo de execução de seu método se a seqüência for implementada usando um arranjo? E se for implementada usando uma lista encadeada?

C-2.8 Projete um algoritmo para desenhar uma árvore binária usando quantidades calculadas a partir de um caminhamento sobre a árvore.

C-2.9 Projete algoritmos para as seguintes operações sobre um nodo v em uma árvore binária T:

- preorderNext(v): retorna o nodo visitado depois de v em um caminhamento prefixado sobre T.
- inorderNext(v): retorna o nodo visitado depois de v em um caminhamento interfixado sobre T.
- postorderNext(v): retorna o nodo visitado depois de v em um caminhamento pós-fixado sobre T.

Quais são os piores casos em relação ao tempo de execução de seus algoritmos?

C-2.10 Forneça um algoritmo que consome tempo $O(n)$ para calcular a profundidade de todos os nodos de uma árvore T, onde n é o número de nodos de T.

C-2.11 O ***fator de balanceamento*** de um nodo interno v de uma árvore binária é a diferença entre as alturas das subárvores direita e esquerda de v. Mostre como especializar o caminhamento de Euler para imprimir os fatores de balanceamento de todos os nodos de uma árvore binária.

C-2.12 Duas árvores ordenadas T' e T'' são ditas ***isomórficas*** se uma das seguintes regras se aplicam:

- T' e T'' consistem em um único nodo.

- Ambas têm o mesmo número k de subárvores, e a i-ésima subárvore T' é isomorfica à i-ésima subárvore de T'', para $i = 1, ..., k$.

Projete um algoritmo que teste quando duas determinadas árvores ordenadas são isomórficas.
Qual o tempo de execução de seu algoritmo?

C-2.13 Seja a ação de visita do caminhamento de Euler indicada por um par (v,a), onde v é o nodo visitado e a é ***direita***, ***abaixo*** *ou* ***esquerda***. Projete um algoritmo para executar a operação tourNext(v,a), que retorna a ação de visita (w,b) que segue (v,a). Qual o pior caso para o tempo de execução de seu algoritmo.

C-2.14 Mostre como representar uma árvore binária imprópria por meio de uma árvore própria.

C-2.15 Seja T uma árvore binária com n nodos. Defina um ***nodo romano*** como sendo um nodo v em T, de forma que o número de descendentes na subárvore esquerda difira do número de descendentes na subárvore direita em no máximo 5. Descreva um método de tempo linear para encontrar cada nodo v de T, de forma que v não seja um nodo romano, mas que todos os seus descendentes o sejam.

C-2.16 Usando pseudocódigo, descreva um método não-recursivo para executar o caminhamento de Euler sobre uma árvore binária que execute em tempo linear e não use uma pilha.

Dica: você pode saber qual a ação que deve ser executada em determinado nodo anotando de onde você vem.

C-2.17 Usando pseudocódigo, descreva um método não-recursivo para executar o caminhamento interfixado sobre uma árvore binária em tempo linear.

C-2.18 Seja T uma árvore binária com n nodos (T pode ou não ser implementada usando um vetor). Forneça um método de tempo linear que use os métodos da interface BinaryTree para percorrer os nodos de T incrementando os valores da função de numeração de níveis p fornecida na Seção 2.3.4. Este caminhamento é conhecido como ***caminhamento por ordem de nível***.

C-2.19 O ***comprimento do caminho*** de uma árvore T é o somatório das profundidades de todos os nodos de T. Descreva um método de tempo linear para calcular o comprimento do caminho de uma árvore T (que não seja necessariamente binária).

C-2.20 Defina o ***comprimento do caminho interno*** $I(T)$ de uma árvore T como sendo a soma das profundidades de todos os nodos internos de T. Da mesma forma, defina o ***comprimento do caminho externo*** $E(T)$ de uma árvore T como sendo a soma das profundidades de todos os nodos externos de T. Mostre que, se T é uma árvore binária como n nodos internos, então $E(T) = I(T) + 2n$.

C-2.21 Seja T uma árvore com n nodos. Defina o ***mais baixo ancestral comum*** (MBAC) entre dois nodos v e w como sendo o nodo mais baixo de T que tenha tanto v como w como descendentes (considerando que um nodo pode

ser descendente de si mesmo). Dados dois nodos v e w, descreva um algoritmo eficiente para encontrar o MBAC entre v e w. Qual o tempo de execução do seu método?

C-2.22 Seja T uma árvore com n nodos e, para qualquer nodo v de T, seja d_v a profundidade de v em T. A ***distância*** entre dois nodos v e w em T é $d_v + d_w - 2d_u$, onde u é o MBAC u de v e w (como definido no exercício anterior). O ***diâmetro*** de T é a distância máxima entre dois nodos de T. Descreva um algoritmo eficiente para encontrar o diâmetro de T. Qual o tempo de execução do seu método?

C-2.23 Suponhamos uma coleção S de n intervalos da forma $[a_i, b_i]$. Projete um algoritmo eficiente para calcular a união de todos os intervalos de S. Qual o tempo de execução do seu método?

C-2.24 Supondo que a entrada de um problema de ordenação seja um vetor A, descreva como implementar um algorimto de ordenação por seleção usando apenas o arranjo A e no máximo seis variáveis adicionais (de tipos básicos).

C-2.25 Supondo que a entrada de um problema de ordenação seja um vetor A, descreva como implementar um algorimto de ordenação por inserção usando apenas o arranjo A e no máximo seis variáveis adicionais (de tipos básicos).

C-2.26 Supondo que a entrada de um problema de ordenação seja um vetor A, descreva como implementar um algorimto de *heap-sort* usando apenas o arranjo A e no máximo seis variáveis adicionais (de tipos básicos).

C-2.27 Suponha que a árvore binária T usada para implementar um *heap* possa ser acessada usando apenas os métodos do TAD árvore binária. Desta forma, podemos assumir que T é implementado como um vetor. Dada uma referência para o último nodo corrente v, descreva um algoritmo eficiente para encontrar o ponto de inserção (isto é, o novo último nodo) usando apenas os métodos da interface árvore binária. Certifique-se de que todos os casos possíveis são tratados. Qual o tempo de execução deste método?

C-2.28 Mostre que para qualquer n existe uma seqüência de inserções em um *heap* que requer tempo $\Omega(n \log n)$ para executar.

C-2.29 Podemos representar o caminho da raiz a um nodo de uma árvore binária usando um *string* binário, onde 0 significa "vá para a esquerda", e 1 significa "vá para a direita". Projete um algoritmo de tempo logarítmico para encontrar o último nodo de um *heap* que armazena n elementos baseado nesta representação.

C-2.30 Mostre que o problema de encontrar o k-ésimo menor elemento em um *heap* leva no mínimo tempo $\Omega(k)$ no pior caso.

C-2.31 Desenvolva um algoritmo para calcular o k-ésimo menor elemento de um conjunto de n inteiros distintos em tempo $O(n + k \log n)$.

C-2.32 Seja T um *heap* que armazena n chaves. Dê um algoritmo eficiente que liste todas as chaves de T que são menores ou iguais a uma dada chave de consulta x (que não se encontra obrigatoriamente em T). Por exemplo, dado o *heap* da Figura 2.41 e a chave de consulta $x = 7$, o algoritmo deve listar 4,5,6 e 7. Observe que as chaves não necessitam ser listadas em ordem. O

ideal é que seu algoritmo execute em tempo $O(k)$, onde k é o número de chaves listadas.

C-2.33 A implementação de um dicionário usando tabela *hash* requer que encontremos um número primo entre M e $2M$. Implemente um método para encontrar tal número primo usando o ***algoritmo da peneira***. Neste algoritmo, alocamos um arranjo A de $2M$ células booleanas, de forma que cada célula i esteja associada com um inteiro i. Inicializamos o arranjo com todas as posições em "true" e "desligamos" todas as células que são múltiplas de 2, 3, 5, 7 e assim por diante. Esse processo pode parar quando encontramos um número maior do que $\sqrt{2M}$.

C-2.34 Dê a descrição em pseudocódigo para executar a remoção de uma tabela *hash* que usa teste linear para resolver colisões em que não precisamos usar um marcador especial para representar elementos deletados. Assim, devemos reorganizar os conteúdos da tabela *hash* de maneira que aparente que os itens removidos nunca foram inseridos.

C-2.35 A estratégia de teste quadrático tem um problema de *cluster* relacionado com a maneira pela qual ele procura por *slots* livres quando uma colisão ocorre. Na verdade, quando uma colisão ocorre em um *bucket* $h(k)$, verificamos $A[h(k) + f(j)) \bmod N]$, para $f(j) = j^2$ usando $j = 1, 2, ..., N - 1$.

a. Mostre que $f(j) \bmod N$ assumirá no máximo $(N + 1)/2$ valores diferentes, para N primo, considerando que j varia de 1 até $N - 1$. Como parte desta justificativa, note que $f(R) = f(N - R)$ para todo R.
b. Uma estratégia melhor é escolher um número primo N de forma que N seja congruente a 2 módulo 4 e então verificar os *buckets* $A[h(k) \pm j^2) \bmod N]$ já que j varia de 1 até $(N - 1)$, alternando entre adição e subtração. Mostre que esta forma alternativa do teste quadrático garante a verificação de todos os *buckets* de A.

Projetos

P-2.1 Escreva um programa que receba como entrada uma expressão aritmética completamente entre parênteses e a converta em uma árvore binária. Seu programa deve mostrar a árvore de alguma forma e também imprimir o valor associado com a raiz. Como desafio adicional, permita que as folhas armazenem variáveis da forma x_1, x_2, x_3 e assim por diante, inicializadas com 0 e permitindo que possam ser alteradas interativamente em seu programa, atualizando a impressão do valor da expressão representada na árvore.

P-2.2 Escreva um programa executável ou uma *applet* que anime um *heap*. O programa deve suportar todas as operações de uma fila de prioridades e deve visualizar as operações de ordenação *up-heap* e *down-heap*. (Extra: visualize também a construção *botom-up* do *heap*).

P-2.3 Execute uma análise comparativa que estude as taxas de colisão para vários códigos *hash* de *string* de caractere, tais como códigos *hash* polinomiais para diferentes valores do parâmetro a. Use uma tabela *hash* para determinar colisões, mas apenas conte as colisões em que diferentes *strings* mapeiam o

mesmo código *hash* (não a mesma posição na tabela *hash*). Teste estes códigos *hash* em arquivos texto encontrados na Internet.

P-2.4 Execute uma análise comparativa como a do exercício anterior, mas para números de telefone de 10 dígitos, em vez de *strings* de caractere.

Notas

Estruturas de dados básicas tais como as pilhas, filas e listas encadeadas apresentadas neste capítulo pertencem ao folclore da ciência da computação. Essas estruturas foram descritas pela primeira vez por Knuth em seu livro original sobre *Fundamental Algorithms* [117]. Neste capítulo, a abordagem utilizada foi definir primeiramente as estruturas básicas de pilha, fila e deque sob a forma de TADs, e então apresentar as implementações concretas. Esta abordagem de especificação e implementação de estruturas de dados segue os avanços da engenharia de *software* a partir da abordagem orientada a objetos, e atualmente é considerada como sendo a abordagem-padrão para o ensino de estruturas de dados. Essa abordagem foi introduzida inicialmente nos livros clássicos de Aho, Hopcroft e Ullmann sobre estruturas de dados e algoritmos [7,8]. Para mais informações sobre tipos abstratos de dados, veja o livro de Liskov e Guttag [135], a revisão do artigo de Cardelli e Wegner [44] ou o capítulo de livro de Demurjian [57]. As convenções de nomenclatura utilizadas para os métodos dos TADs pilhas, filas e deques foram tirados de JDSL [86]. JDSL é uma biblioteca de estruturas de dados em Java construída usando uma abordagem extraída das bibliotecas C++ STL[158] e LEDA [151]. Usaremos esta convenção ao longo de todo este texto. Neste capítulo, motivamos o estudo de pilhas e filas a partir de questões de implementação em Java. Ao leitor que deseja saber mais sobre o ambiente de execução de Java, conhecido como a Máquina Virtual Java (Java Virtual Machine – JVM), indicamos o livro de Lindholm e Yellin [134] que define a JVM.

Seqüências e iteradores são conceitos difundidos nas bibliotecas padrão de C++ (STL) [158], e definem regras fundamentais na JDSL, a biblioteca de estruturas de dados de Java. O TAD seqüência é uma extensão genérica do java.util.Vector da API de Java (por exemplo, veja o livro de Arnold e Gosling [13]), e o TAD lista é proposto por vários autores incluindo Aho, Hopcroft e Ullmann [8], que introduz a abstração "posição", e Wood [211], que define um TAD lista similar ao nosso. Implementações de seqüências usando arranjos e listas encadeadas são discutidos no livro original de Knuth, *Fundamental Algorithms* [118]. O livro dos companheiros de Knuth, *Sorting and Searching* [119], descreve o método de ordenação da bolha e a história deste e de outros algoritmos.

A idéia de enxergar estruturas de dados como contêineres (e outros princípios de projeto orientado a objetos) pode ser encontrada em livros de projeto orientado a objetos tais como o de Booch [32] e Budd [42]. O conceito também existe sob o nome de "coleção de classes" nos livros de Golberg e Robson [79] e Liskov e Guttag [135]. Nosso uso da abstração "posição" deriva das abstrações "posição" e "nodo" introduzidas por Aho, Hopcroft e Ullmann [8]. Discussões a respeito dos caminhamentos clássicos prefixado, interfixado e pós-fixado podem ser encontrados no livro do Knuth,

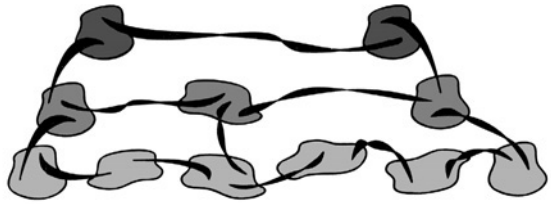
Fundamental Algorithms [118]. A técnica do caminhamento de Euler é originária da comunidade de processamento paralelo, e foi introduzida por Tarjan e Vishkin [197] e é discutida por JáJá [107] e por Karp e Ramachandran [114]. O algoritmo para desenhar uma árvore é, em geral, considerado como parte do floclore dos algoritmos de desenho de grafos. Ao leitor interessado no desenho de grafos, indicamos os trabalhos de Tamassia [194] e Di Battista et al. [58, 59]. O quebra-cabeças do Exercício R-2.2 foi trazido por Micha Sharir.

O livro de Knuth sobre ordenação e pesquisa [119] descreve a motivação e a história dos algoritmos *selection-sort*, *insertion-sort* e *heap-sort*. O algoritmo *heap-sort* é atribuído a Williams [210], e a construção de tempo linear para o *heap* é atribuída a Floyd [70]. Algoritmos adicionais e análises de variações do *heap* e do *heap-sort* podem ser encontrados nos artigos de Bentley [29], Carlsson [45], Gonnet e Munro [82], McDiarmid e Reed [141] e Schaffer e Sedgewick [178]. O padrão localizador (também descrito em [86]) parece ser novo.

Capítulo

Árvores de pesquisa e *skip lists*

3



3.1 Dicionários ordenados e árvores binárias 149

3.1.1 Tabelas ordenadas 149

3.1.2 Árvores binárias de pesquisa 152

3.1.3 Pesquisa em árvore binária. 153

3.1.4 Inserção em uma árvore de pesquisa binária 155

3.1.5 Remoção em uma árvore de pesquisa binária. 156

3.1.6 Desempenho de árvores de pesquisa binárias 158

3.2 Árvores AVL. 159

3.2.1 Operações de atualização 160

3.2.2 Desempenho 165

3.3 Árvores de pesquisa com profundidade limitada 165

3.3.1 Árvores de pesquisa genéricas. 166

3.3.2 Árvores (2,4). 169

3.3.3 Árvores vermelho-pretas. 175

3.4 *Splay trees*. 187

3.4.1 Espalhamento 189

3.4.2 Análise amortizada do espalhamento 194

3.5 *Skip lists*. 198

3.5.1 Pesquisa 200

3.5.2 Operações de atualização 201

3.5.3 Análise probabilística das *skip lists* 203

3.6 Exemplo em Java: árvores AVL e árvores vermelho-pretas 205

3.6.1 Implementação Java de árvores AVL. 209

3.6.2 Implementação Java de árvores vermelho-pretas. 212

3.7 Exercícios 215

As pessoas gostam de escolher. Apreciamos dispor de maneiras diversas de resolver um mesmo problema, pois assim podemos confrontar diferentes pontos fortes e fracos. Este capítulo é dedicado a explorar diferentes formas de implementar um dicionário ordenado. Iniciaremos este capítulo discutindo árvores binárias e como elas suportam uma implementação simples baseada em árvores de um dicionário ordenado, mas não garantem eficiência no pior caso. Entretanto, elas formam a base da maioria das implementações de dicionário baseadas em árvores, e muitas serão analisadas neste capítulo. Uma das implementações clássicas é a árvore AVL, apresentada na Seção 3.2, que corresponde a uma árvore binária que obtém tempo logarítmico de busca para operações de pesquisa e atualização.

Na Seção 3.3, introduzimos o conceito de árvore de profundidade limitada, que mantém todos os nodos externos na mesma profundidade ou "pseudoprofundidade". Uma árvore deste tipo é a árvore genérica, que é uma árvore ordenada em que cada nodo interno pode armazenar vários itens e ter vários filhos. Uma árvore genérica é uma generalização da árvore binária e, da mesma forma que uma árvore binária, pode ser especializada em uma estrutura eficiente para dicionários ordenados. Um tipo específico de árvore genérica discutido na Seção 3.3 é a árvore (2-4), que é uma árvore de profundidade limitada na qual cada nodo interno armazena uma, duas ou três chaves e tem dois, três ou quatro filhos, respectivamente. A vantagem destas árvores é que elas têm algoritmos para a inserção e remoção de chaves que são simples e intuitivos. Operações de atualização reorganizam árvores (2-4) usando operações naturais que dividem e combinam nodos próximos ou transferem chaves entre os mesmos. Uma árvore (2-4) que armazena n itens ocupa um espaço $O(n)$ e suporta pesquisas, inserções e remoções em tempo $O(\log n)$ no pior caso. Outro tipo de árvore de profundidade limitada estudada nesta seção é a árvore vermelho-preta. Estas são árvores binárias cujos nodos são coloridos de "vermelho" e "preto", de forma que o esquema de cores garanta que cada nodo externo esteja na mesma "profundidade preta" (logarítmica). A noção de pseudoprofundidade resulta de uma correspondência entre árvores vermelho-pretas e árvores (2-4). Usando essa correspondência, motivamos e tornamos intuitivos alguns algoritmos complexos de inserção e remoção em árvores vermelho-pretas, que são baseados em rotações e troca de cores. Uma vantagem obtida pelas árvores vermelho-pretas sobre os demais tipos de árvores binárias (como as árvores AVL) é que elas podem ser reestruturadas após uma inserção ou remoção com apenas $O(1)$ rotações.

Na Seção 3.4, discutimos *splay trees*, que chamam a atenção pela simplicidade de seus métodos de pesquisa e atualização. *Splay trees* são árvores binárias que, após cada pesquisa, inserção ou deleção, movem o nodo acessado em direção à raiz através de uma seqüência cuidadosamente coreografada de rotações. Essa heurística simples de "mover para cima", auxilia essa estrutura de dados a se adaptar aos tipos de operações sendo executadas. Um dos resultados dessa heurística é que *splay trees* garantem que o tempo amortizado de execução de cada operação do dicionário seja logarítmico.

Finalmente, na Seção 3.5, discutimos listas de *skip*, que não são árvores, mas usam a mesma noção de profundidade que mantém todos os elementos em uma profundidade logarítmica. Essas estruturas, entretanto, são aleatórias, de maneira que os limites de profundidade são probabilísticos. Em especial, mostramos que é muito provável que a altura de uma *skip lists* que armazena n elementos é $O(\log n)$. Admite-se que isso não é tão forte quanto um verdadeiro pior caso de limite, mas as operações de atualização para *skip lists* são simples, e as comparações são favoráveis às árvores na prática.

Focamos a implementação prática de árvores binárias na Seção 3.6, apresentando implementações Java tanto para árvores AVL como para árvores vermelho-pretas. Destacamos como essas duas estruturas de dados podem ser construídas a partir do TAD árvore discutido na Seção 2.3.

Reconhecemos que são poucos os tipos de estruturas de dados discutidos neste capítulo e que o leitor ou instrutor com pouco tempo pode estar interessado apenas em estudar tópicos selecionados. Por esta razão, projetamos este capítulo de forma que cada seção possa ser estudada de forma independente das demais, exceto a primeira, que é apresentada em seguida.

3 .1 Dicionários ordenados e árvores binárias

Em um dicionário ordenado, desejamos realizar as operações usuais de um dicionário, discutidas na Seção 2.5.1, tais como findElement(k), insertItem(k, e) e removeElement(k), mas também manter a relação de ordem entre as chaves. Podemos usar um comparador para fornecer a relação de ordem entre as chaves e, como veremos, o uso de uma ordem facilita a tarefa de implementar eficientemente o TAD dicionário. Além disso, o dicionário ordenado também suporta os seguintes métodos, além dos incluídos no TAD dicionário:

closestKeyBefore(k): retorna a chave do item com a maior chave menor ou igual a k.

closestElemBefore(k): retorna o elemento do item com a maior chave menor ou igual a k.

closestKeyAfter(k): retorna a chave do item com a menor chave maior ou igual a k.

closestElemAfter(k): retorna o elemento do item com a menor chave maior ou igual a k.

Cada um destes métodos retorna o objeto especial NO_SUCH_KEY se nenhum item no dicionário satisfizer a consulta.

A natureza ordenada das operações acima faz com que o uso de um arquivo de *log* ou tabela *hash* seja inapropriado para implementar o dicionário, pois nenhuma dessas estruturas de dados mantém qualquer informação de ordenação das chaves no dicionário. De fato, tabelas *hash* têm seu melhor desempenho quando as chaves estão distribuídas o mais aleatoriamente possível. Assim, devemos considerar novas implementações para dicionários quando desejamos trabalhar com dicionários ordenados.

Tendo definido o tipo abstrato de dados dicionário, vamos considerar então formas simples de implementar este TAD.

3.1.1 Tabelas ordenadas

Se um dicionário D é ordenado, podemos armazenar seus itens em um vetor S em ordem não-decrescente de chave. Especificamos que S é um vetor e não uma seqüência genérica, pois a ordenação das chaves no vetor S permite uma pesquisa mais rápida do que seria possível se S fosse, por exemplo, uma lista encadeada. Referimo-nos a essa

implementação de um dicionário D com vetor ordenado como uma ***tabela de pesquisa***. Essa implementação contrasta com uma implementação baseada em arquivos de *log*, que usa uma seqüência não-ordenada para implementar o dicionário.

A necessidade de espaço para uma tabela de pesquisa é $\Theta(n)$, similar ao arquivo de *log*, supondo que podemos aumentar e diminuir o tamanho do arranjo implementando o vetor S para manter o tamanho deste arranjo proporcional ao número de itens no vetor. Diferentemente de um arquivo de *log*, fazer atualizações em uma tabela de pesquisa toma um tempo considerável. Em particular, realizar a operação insertItem(k,e) em uma tabela de pesquisa requer tempo $O(n)$ no pior caso, já que precisamos mover para a frente todos os itens no dicionário que têm chave maior do que k para abrir espaço para o novo item (k,e). A implementação com tabela de pesquisa é, portanto, inferior ao arquivo de *log* em termos de tempos de execução de pior caso para as operações de atualização de dicionários. Mesmo assim, podemos efetuar as operações findElement e findAllElements muito mais rapidamente em uma tabela de pesquisa.

Pesquisa binária

Uma vantagem significativa do uso de um vetor S baseado em um arranjo para implementar um dicionário ordenado D com n itens é que acessar um elemento de S através de sua ***colocação*** toma tempo $O(1)$. Lembramos da Seção 2.2.1 que a colocação de um elemento em um vetor é o número de elementos que o precedem. Assim, o primeiro elemento em S tem colocação 0, e o último elemento tem colocação $n-1$.

Os elementos em S são os itens do dicionário D e, já que S é ordenado, o item com colocação i tem uma chave que não é menor do que as chaves dos itens com colocações $0, 1,..., i-1$ e não é maior do que as chaves dos itens com colocações $i+1,..., n-1$. Essa observação permite que achemos um item usando um método de procura muito rapidamente. Chamamos de ***candidato*** um item I de D se, no estágio atual de procura, não podemos garantir que a chave de I é diferente de k. O algoritmo mantém dois parâmetros, low e high, tais que todos os itens candidatos em S têm colocação maior ou igual a low e menor ou igual a high. Inicialmente, temos low = 0 e high = $n-1$, e fazemos key(i) identificar a chave na colocação i, que tem elem(i) como seu elemento. Então comparamos k à chave do candidato mediano, ou seja, o item com colocação

$$\text{mid} = \lfloor(\text{low} + \text{high})/2\rfloor.$$

Consideramos três casos:

- Se k = key(mid), então achamos o item que estávamos procurando, e a pesquisa termina com sucesso, retornando elem(mid).
- Se $k <$ key(mid), então reexaminamos a primeira metade do vetor, ou seja, a metade com colocações entre low e mid – 1.
- Se $k >$ key(mid), então reexaminamos a segunda metade do vetor, ou seja, a metade com colocações entre mid + 1 e high.

Esse método de pesquisa é chamado de ***pesquisa binária*** e é apresentado em pseudocódigo no Algoritmo 3.1. A operação findElement(k) em um dicionário de n itens implementado como um vetor S consiste na chamada de BinarySearch(S,k,0,$n-1$).

Algortimo BinarySearch(S, k, low, high):
  Entrada: um vetor ordenado S armazenando n itens cujas chaves são acessadas com o método key(i) e cujos elementos são acessados com o método element(i); uma chave k para pesquisa; inteiros low e high.
  Saída: um elemento de S com chave k e colocação entre low e high se um tal elemento existir, ou o elemento especial NO_SUCH_KEY
  se low > high **então**
    retorne NO_SUCH_KEY
  senão
    mid $\leftarrow \lfloor$(low + high)/2$\rfloor$
    se k = key(mid) **então**
      retorne elem(mid)
    senão se k < key(mid) **então**
      retorne BinarySearch(S, k, low, mid – 1)
    senão
      retorne BinarySearch(S, k, mid + 1, high)

Algoritmo 3.1 Pesquisa binária em um vetor ordenado.

Ilustramos o algoritmo de pesquisa binária na Figura 3.2.

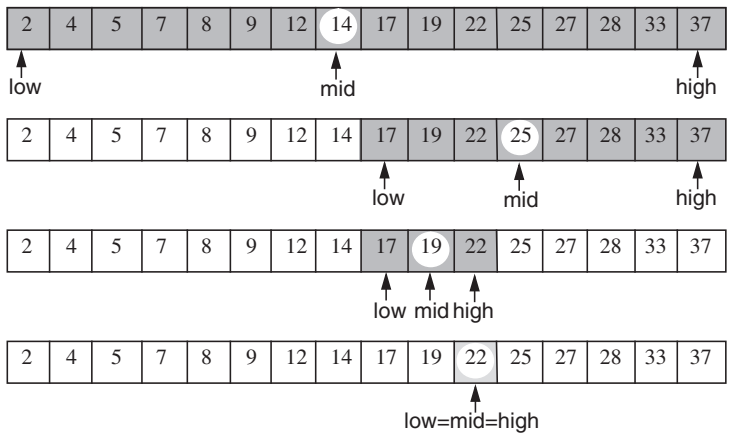


Figura 3.2 Exemplo de pesquisa binária para realizar a operação findElement(22), em um dicionário com chaves inteiras implementado com um vetor ordenado baseado em arranjo. Para manter a simplicidade, mostramos as chaves armazenadas no dicionário, e não os elementos.

Considerando o tempo de execução da pesquisa binária, observamos que um número constante de operações primitivas é executado a cada chamada recursiva do método BinarySearch. Portanto, o tempo de execução é proporcional ao número de chamadas recursivas realizado. Um fato crucial é que com cada chamada o número de candidatos que ainda devem ser examinados em S é dado pelo valor high – low + 1. Além disso, o número de candidatos restantes é reduzido no mínimo pela metade a cada chamada. Especificamente, pela definição de mid, o número de candidatos restantes é

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\le\frac{\text{high}-\text{low}+1}{2}$$

ou

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}.$$

Inicialmente o número de itens candidatos é n; após a primeira chamada a BinarySearch, ele é de no máximo $n/2$; após a segunda, ele é de no máximo $n/4$, e assim sucessivamente. Assim, se representamos o tempo de execução deste método por uma função $T(n)$, então podemos caracterizar o tempo de execução do algoritmo recursivo de pesquisa binária como

$$T(n) \leq \begin{cases} b & \text{se } n < 2 \\ T(n/2) + b & \text{senão,} \end{cases}$$

onde b é constante. Em geral, essa equação recursiva mostra que o número de itens candidatos remanescentes após cada chamada recursiva é de no máximo $n/2^i$. (Equações recursivas como estas são discutidas em mais detalhes na Seção 5.2.1.) No pior caso (elemento não existe), as chamadas recursivas param quando não há mais itens candidatos. Portanto, o número máximo de chamadas recursivas feitas é o menor inteiro m tal que $n/2^m < 1$. Em outras palavras (e lembrando que omitimos a base de um logaritmo quando esta é 2), $m > \log n$. Assim, temos $m = \lfloor \log n \rfloor + 1$, o que implica que BinarySearch(S, k, 0, $n - 1$) e também o método findElement são executados em tempo $O(\log n)$.

A Tabela 3.3 compara os tempos de execução dos métodos de um dicionário implementado usando um arquivo de *log* ou tabela de pesquisa. Um arquivo de *log* permite inserções rápidas mas com pesquisas e remoções lentas, enquanto que a tabela de pesquisa permite pesquisas rápidas mas com inserções e remoções lentas.

Método	Arquivo de *log*	Tabela de pesquisa
findElement	$O(n)$	$O(\log n)$
insertItem	$O(1)$	$O(n)$
removeElement	$O(n)$	$O(n)$
closestKeyBefore	$O(n)$	$O(\log n)$

Tabela 3.3 Comparação dos tempos de execução dos principais métodos de um dicionário implementado usando um arquivo de *log* ou de uma tabela de pesquisa. Indicamos o número de itens no dicionário quando a operação é executada com n. O desempenho dos métodos closestElemBefore, closestKeyAfter e closestElemAfter é similar ao de closestKeyBefore.

3.1.2 Árvores binárias de pesquisa

A estrutura de dados que será apresentada nesta seção, a árvore binária de pesquisa, aplica a motivação de um algoritmo de pesquisa binária a uma estrutura de dados baseada em árvore. Definimos uma árvore binária de pesquisa como sendo uma árvore binária na qual cada nodo interno v armazena um elemento e de forma que os elementos armazenados na subárvore esquerda de v sejam menores ou iguais a e, e os elementos armazenados na subárvore direita de v sejam maiores ou iguais a e. Além disso, vamos assumir que os nodos externos não armazenam elementos; desta forma, eles podem ser nulos ou referências para o objeto NULL_NODE.

Um caminhamento interfixado de uma árvore de pesquisa binária visita os elementos armazenados nesta árvore em ordem não-decrescente. Uma árvore de pesquisa binária suporta pesquisa, desde que a pergunta feita a cada nodo interno seja se o elemento armazenado no nodo é maior, menor ou igual ao elemento sendo pesquisado.

Podemos usar uma árvore de pesquisa binária T para localizar um elemento com um certo valor x caminhando pela árvore T. A cada nodo interno, comparamos o valor do nodo corrente com o elemento x sendo pesquisado. Se a resposta à comparação for "menor", então a pesquisa continua na subárvore esquerda. Se a resposta for "igual", então a pesquisa se encerra com sucesso. Se a resposta for "maior", então a pesquisa continua na subárvore direita. Finalmente, se for atingido um nodo externo (que é um nodo vazio), então a pesquisa termina sem sucesso (veja a Figura 3.4).

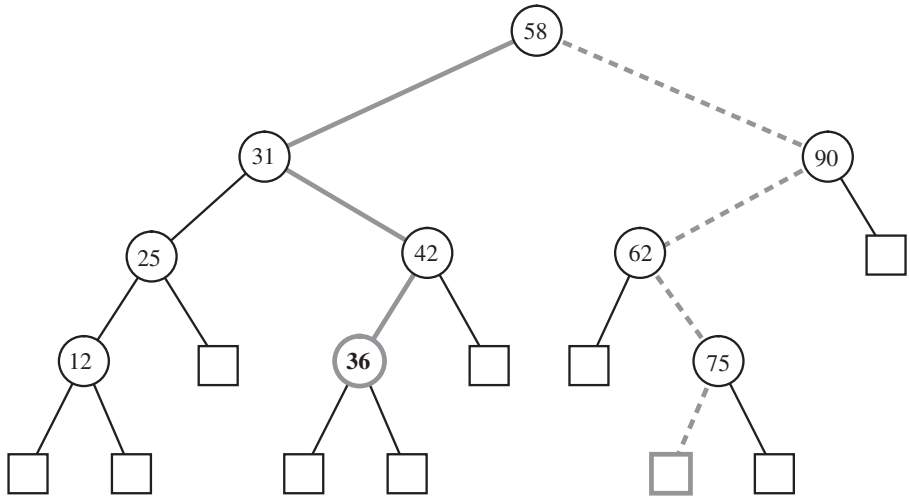


Figura 3.4 Árvore de pesquisa binária que armazena números inteiros. As linhas sólidas grossas indicam o caminho percorrido na pesquisa bem-sucedida pelo número 36. A linha grossa tracejada indica o caminho percorrido na pesquisa (sem sucesso) pelo número 70.

3.1.3 Pesquisa em árvore binária

Formalmente, uma ***árvore binária de pesquisa*** é uma árvore binária T cujos nodos internos v de T armazenam um item (k, e) de um dicionário D, e as chaves armazenadas na subárvore esquerda de v são menores ou iguais a k, enquanto que as chaves armazenadas na subárvore direita de v são maiores ou iguais a k.

No Algoritmo 3.5, apresentamos o método recursivo TreeSearch, baseado na estratégia de pesquisa em uma árvore binária T acima. Dada uma chave de pesquisa k e um nodo v *de* T, o método TreeSearch retorna um nodo (posição) w pertencente à subárvore $T(v)$ de T enraizada em v, de forma que um dos seguintes dois casos ocorre:

- w é o nodo interno de $T(v)$ que armazena a chave k;
- w é um nodo externo de $T(v)$, e todos os nodos interno de $T(v)$ que precedem w no caminhamento interfixado têm chaves menores do que k e todos os nodos internos de $T(v)$ que seguem w no caminhamento interfixado têm chaves maiores do que k.

Deste modo, o método findElement(k) pode ser executado em um dicionário D chamando-se o método TreeSearch(k,T.root()) sobre T. Seja w o nodo de T retornado por esta chamada ao método TreeSearch. Se o nodo w for interno, retornamos o elemento armazenado em w; por outro lado, se w for externo, então retornamos NO_SUCH_KEY.

Algoritmo TreeSearch(k,v):
  ***Entrada*:** uma chave de pesquisa k e um nodo v de uma árvore binária de pesquisa T
  ***Saída*:** um nodo w de uma subárvore $T(v)$ de T enraizada em v, de maneira que w é tanto um nodo interno que armazena a chave k ou w é o nodo externo ao qual o item com chave k pertenceria se existisse.

```
se v é um nodo externo então
  retorne v
se k = key(v) então
  retorne v
senão se k < key(v) então
  retorne TreeSearch(k,T.leftChild(v))
senão
  {sabemos que k > key(v)}
  retorne TreeSearch(k,T.rigthChild(v))
```

Algoritmo 3.5 Pesquisa binária recursiva sobre uma árvore de pesquisa binária.

Observe que o tempo de execução de uma pesquisa sobre uma árvore binária de pesquisa T é proporcional à altura de T. Uma vez que a altura de uma árvore T com n nodos pode ser pequena como $O(\log n)$ ou grande como $\Omega(\log n)$, árvores binárias de pesquisa são mais eficientes quando elas tiverem pouca altura.

Análise da árvore de pesquisa binária

A análise formal do pior caso para o tempo de execução de uma pesquisa em uma árvore de pesquisa binária T é simples. O algoritmo de pesquisa em árvore binária executa um número constante de operações primitivas em cada um dos nodos percorridos sobre a árvore. Cada novo passo do caminhamento é feito sobre um filho do nodo anterior. Isto é, o algoritmo de pesquisa sobre árvore binária é executado nos nodos de um caminho de T que inicia na raiz e desce um nível por vez. Assim, o número de nodos é limitado por $h + 1$, onde h é a altura de T. Em outras palavras, uma vez que gastarmos um tempo $O(1)$ em cada nodo encontrado na pesquisa, o método findElement (ou qualquer outra operação-padrão de pesquisa) executa em tempo $O(h)$, onde h é a altura da árvore de pesquisa binária T usada para implementar o dicionário D (veja a Figura 3.6).

Podemos demonstrar também que uma variação do algoritmo acima executa a operação findAllElements(k), que encontra todos os itens do dicionário com chave k em tempo $O(h + s)$, onde s é o número de elementos retornados. Entretanto, esse método é um pouco mais complicado, e seu detalhes ficam como exercício (C-3.3).

Na verdade, a altura h de T pode ser tão grande quanto n, mas esperamos que normalmente seja menor. Além disso, mostraremos nas próximas seções deste capítulo como manter o limite superior de $O(\log n)$ usando a altura da árvore de pesquisa T. Antes de apresentarmos tal esquema, entretanto, descreveremos implementações de método de atualização de dicionários em uma árvore binária possivelmente desbalanceada.

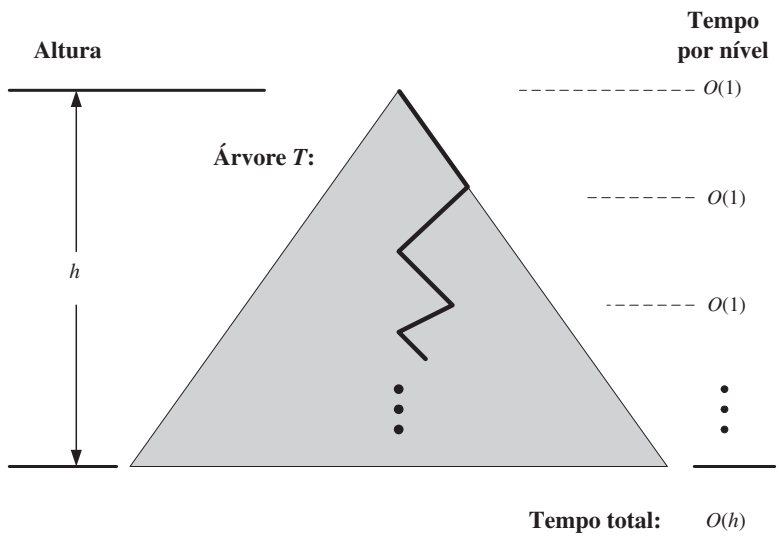


Figura 3.6 Demonstração o tempo de execução de uma pesquisa sobre uma árvore binária de pesquisa. A figura usa as formas de representação-padrão visualizando uma árvore binária de pesquisa como um grande triângulo e o caminho a partir da raiz como um linha em ziguezague.

3.1.4 Inserção em uma árvore de pesquisa binária

Árvores de pesquisa binárias permitem implementar as operações insertItem e removeElement usando algoritmos que são mais diretos, mas não triviais.

Para executar a operação insertItem(k,e) em um dicionário D implementado com uma árvore de pesquisa binária T, iniciamos chamando o método TreeSearch(k,T.root()) sobre T. Seja w o nodo retornado por TreeSearch.

- Se w for um nodo externo (nenhum item com a chave k está armazenado em T), substituímos w por um novo nodo interno que armazena o item (k, e) e dois filhos externos, da mesma forma que a operação expandExternal(w) sobre T (veja a Seção 2.3.3). Notamos que w é o local apropriado para inserir o item com chave k.
- Se w for um nodo interno (outro item com chave k está armazenado em w), chamamos TreeSearch(k,rightChild(w)) (ou, de forma equivalente, Tree-Search(k,leftChild(w))) e recursivamente aplicamos o algoritmo ao nodo retornado por TreeSearch.

O algoritmo de inserção acima, por fim, percorre o caminho da raiz de T descendo até um nodo externo, que é substituído por um novo nodo interno, acomodando o novo item. Conseqüentemente, uma inserção adiciona o novo item "na parte de baixo" da árvore de pesquisa T. Um exemplo de inserção em uma árvore de pesquisa binária é apresentado na Figura 3.7.

A análise do algoritmo de inserção é análoga ao da pesquisa. O número de nodos visitados é proporcional à altura h de T, no pior caso. Além disso, supondo uma implementação usando estrutura encadeada para T (veja a Seção 2.3.4), gastamos tempo $O(1)$ em cada nodo visitado. Desta forma, o método insertItem executa em tempo $O(h)$.

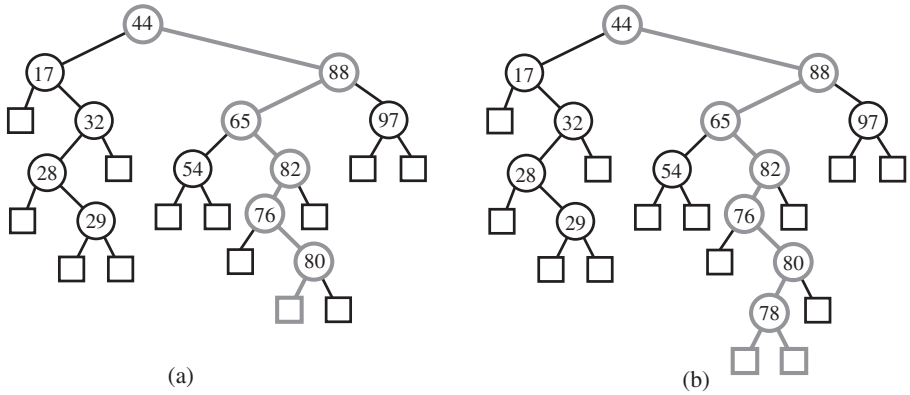


Figura 3.7 Inserção de um item com chave 78 em uma árvore de pesquisa binária. A determinação da posição de inserção é mostrada em (a), e a árvore resultante é apresentada em (b).

3.1.5 Remoção em uma árvore de pesquisa binária

Executar a operação removeElement(k) sobre um dicionário D implementado usando uma árvore de pesquisa binária T é um pouco mais complexo, uma vez que não queremos criar "buracos" na árvore T. Este buraco, onde um nodo interno não armazena um elemento, dificulta, senão impossibilita, executar pesquisas corretamente em uma árvore de pesquisa binária. De fato, se ocorrerem várias remoções que não se preocupem em reorganizar a árvore T, poderá surgir uma grande seção onde nodos internos não armazenam elementos, o que pode confundir pesquisas futuras.

A operação de remoção começa de forma simples, uma vez que iniciamos executando o algoritmo TreeSearch(k,T.root()) sobre T, de maneira a localizar o nodo que armazena a chave k. Se TreeSearch retorna um nodo externo, então não existe um elemento com chave k no dicionário D, e retornamos o elemento especial NO_SUCH_KEY e pronto. Se, por outro lado, TreeSearch retorna um nodo interno w, então w armazena um item que desejamos remover.

Distinguimos dois casos (de complexidade crescente) sobre como proceder, baseado em quanto o nodo w é fácil de resolver ou não:

- Se um dos filhos do nodo w é um nodo externo, digamos z, simplesmente removemos w e z de T, usando a operação removeAboveExternal(z) sobre T. Esta operação (veja a Figura 2.26 e Seção 2.3.4) reestrutura T, substituindo w pelo irmão de z, removendo tanto w como z de T.

Esse caso é demonstrado na Figura 3.8.

- Se os dois filhos do nodo w são nodos internos, não podemos simplesmente remover o nodo w de T, uma vez que isso criaria um "buraco" em T. Em vez disso, procedemos como segue (veja a Figura 3.9):

  1. Encontramos o primeiro nodo interno y que segue w em um caminhamento interfixado sobre T. O nodo y é o nodo mais interno à esquerda na subárvore direita de w e é encontrado visitando primeiramente o filho da direita de w e, então, descendo a árvore T, seguindo o filho da esquerda.

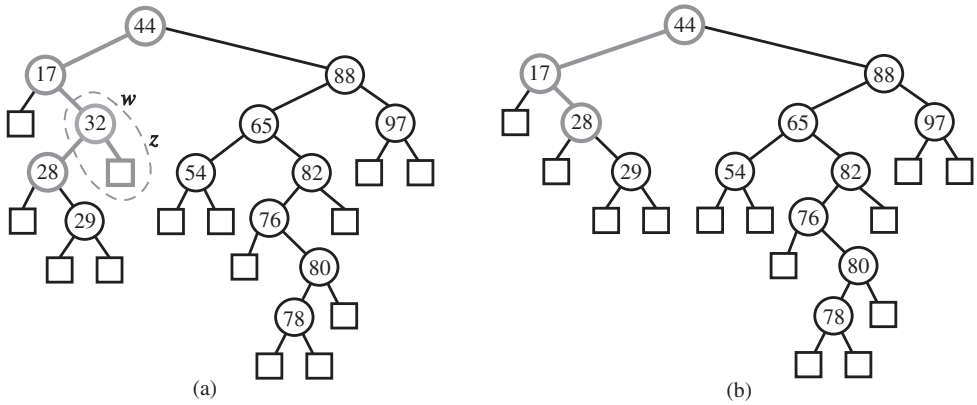


Figura 3.8 Remoção da árvore de pesquisa binária da Figura 3.7b, onde a chave a ser removida (32) está armazenada em um nodo *w* com um filho externo: (a) mostra a árvore antes da remoção, juntamente com os nodos afetados pela operação removeAboveExternal(*z*) sobre *T*; (b) apresenta a árvore depois da remoção.

Também, o filho da esquerda *x* de *y* é o nodo externo que segue imediatamente o nodo *w,* em um caminhamento interfixado de *T*.

2. Salvamos o elemento armazenado em *w* em uma variável temporária *t* e movemos o item de *y* em *w*. Essa ação tem o efeito de remover o primeiro item armazenado em *w*.
3. Removemos os nodos *x* e *y* de *T* usando a operação removeAboveExternal(*x*) sobre *T*. Essa ação substitui *y* pelo irmão de *x* e remove *x* e *y* de *T*.
4. Retornamos o elemento previamente armazenado em *w*, que salvamos na variável temporária *t*.

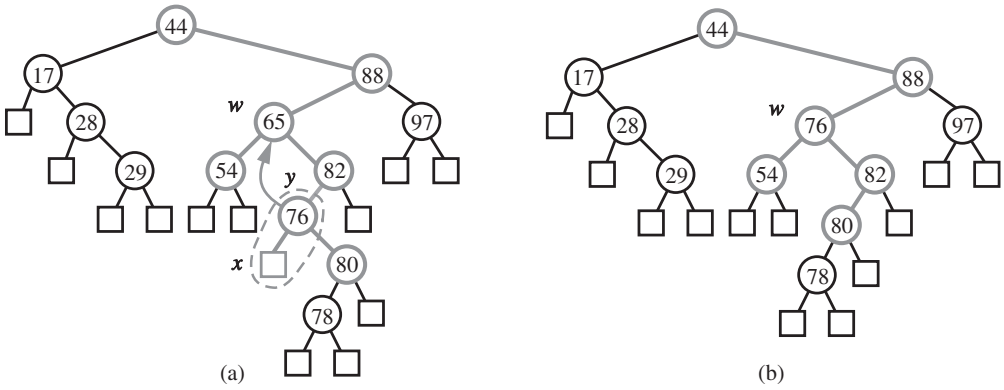


Figura 3.9 Remoção da árvore de pesquisa binária da Figura 3.7b, onde a chave a ser removida (65) está armazenada em um nodo cujos filhos são ambos internos: (a) antes da remoção; (b) depois da remoção.

A análise do algoritmo de remoção é análoga a dos algoritmos de inserção e pesquisa. Gastamos tempo $O(1)$ em cada nodo visitado e, no pior caso, o número de nodos visitados é proporcional à altura *h* de *T*. Portanto, em um dicionário *D* implemen-

tado usando uma árvore de pesquisa binária T, o método removeElement executa em tempo $O(h)$, onde h é a altura da árvore T.

Podemos mostrar também uma variação do algoritmo apresentado que executa a operação removeAllElements(k) em tempo $O(h + s)$, onde s é o número de elementos no iterador retornado. Os detalhes são deixados como exercício (C-3.4).

3.1.6 Desempenho de árvores de pesquisa binárias

O desempenho de um dicionário implementado com pesquisa binária é resumida no teorema a seguir e na Tabela 3.10.

Teorema 3.1: *uma árvore de pesquisa binária T com altura h para n itens chave-elemento usa espaço O(n) e executa as operações do TAD dicionário com os seguintes tempos de execução. Operações* size *e* isEmpty *consomem tempo* $O(1)$ *cada. Operações* findElement, insertItem *e* removeElement *consomem tempo O(h) cada uma. Operações* findAllElements *e* removeAllElements *consomem tempo O(h + s) cada, onde s é o tamanho dos iteradores retornados.*

Método	Tempo
size, isEmpty	$O(1)$
findElement, insertItem, removeElement	$O(h)$
findAllElements, removeAllElements	$O(h + s)$

Tabela 3.1 Tempos de execução dos principais métodos de um dicionário implementado usando uma árvore binária de pesquisa. Indicamos com h a altura corrente da árvore e com s o tamanho dos iteradores retornados por findAllElements e removeAllElements. O espaço utilizado é $O(n)$, onde n é o número de itens armazenados no dicionário.

Observamos que o tempo de execução das operações de pesquisa e atualização em uma árvore de pesquisa binária varia muito dependendo da altura da árvore. Apesar de tudo, podemos estar certos de que, na média, uma árvore de pesquisa binária com n chaves, gerada por uma série de inserções e remoções aleatórias de chaves, tem uma altura esperada $O(\log n)$. Tal declaração exige uma linguagem matemática acurada para definir com precisão o queremos dizer com uma série aleatória de inserções e remoções e uma teoria probabilística sofisticada para justificá-la; em conseqüência, essa justificativa está fora do escopo deste livro. Desta forma, nos contentamos em saber quais seqüências aleatórias de atualizações geram árvores de pesquisa binárias que têm altura logarítmica na média, mas tendo em mente que o desempenho é pobre no pior caso, devemos também ser cuidadosos ao usar árvores de pesquisa binárias padrão em aplicações em que as atualizações não são aleatórias.

A relativa simplicidade desta implementação de dicionário, combinada com o bom desempenho do caso médio, torna árvores de pesquisa binárias estruturas de dados atraentes para dicionários em aplicações em que as chaves inseridas e removidas seguem um padrão aleatório e que tempos de resposta mais longos eventuais são aceitáveis. Existem, entretanto, aplicações em que é essencial dispor de um dicionário com tempo de resposta rápido de pesquisa e atualização no pior caso. A estrutura de dados apresentada nas próximas seções atende a este objetivo.

3.2 Árvores AVL

Na seção anterior, discutimos o que deve ser uma estrutura de dados dicionário eficiente, mas o desempenho atingido no pior caso para várias operações é linear em relação ao tempo, o que não é melhor que o de uma implementação de um dicionário baseado em uma seqüência (tal como arquivos de *log* e tabelas de pesquisa). Nesta seção, descrevemos uma forma simples de corrigir este problema de maneira a obter tempo logarítmico para todas as operações fundamentais do dicionário.

Definição

A correção se resume a acrescentar uma regra à definição de árvore binária de pesquisa que irá manter uma altura logarítmica para a árvore. A regra que consideramos nesta seção é a ***propriedade de altura/balanceamento***, que caracteriza a estrutura de uma árvore binária de pesquisa T em função da altura de seus nodos internos (lembre da Seção 2.3.2 que a altura de um nodo v em uma árvore é o comprimento do caminho mais longo partindo de v até um nodo externo):

***Propriedade de altura/balanceamento*:** para cada nodo interno v de T, as alturas dos filhos de v podem variar em no máximo 1.

Qualquer árvore binária de pesquisa T que satisfaça essa propriedade é dita ser uma árvore AVL, que é um conceito cujo nome é tirado das iniciais de seus autores: Adel'son-Vel'skii e Landis. Um exemplo de árvore AVL é apresentado na Figura 3.11.

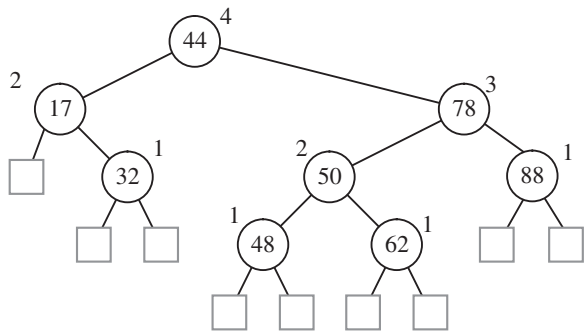


Figura 3.11 Um exemplo de árvore AVL. As chaves são apresentadas dentro dos nodos, e as alturas são mostradas próximas aos mesmos.

Uma conseqüência imediata da propriedade de altura/balanceamento é que uma sub-árvore de uma árvore AVL é também uma árvore AVL. A propriedade de altura/balanceamento é também importante por manter a altura pequena, como mostrado na proposição a seguir.

Teorema 3.2: *a altura de uma árvore AVL T que armazena n itens é $O(\log n)$.*

Prova: em vez de tentar encontrar o limite superior para a altura de uma árvore AVL diretamente, é mais fácil trabalhar no "problema inverso", ou seja, encontrar o limite inferior do número mínimo de nodos internos $n(h)$ de uma árvore AVL com altura h. Mostraremos que $n(h)$ cresce no mínimo exponencialmente, isto é, $n(h)$ é $\Omega(c^h)$ para

uma constante $c > 1$. A partir disso, é simples derivar que a altura de uma árvore AVL que armazena n chaves é $O(\log n)$.

Observamos que $n(1) = 1$ e $n(2) = 2$, porque uma árvore AVL de altura 1 tem apenas um nodo interno, e uma árvore AVL de altura 2 tem dois nodos internos. Agora, para $h \geq 3$, uma árvore AVL com altura h e número de nodos mínimo é tal que suas duas subárvores são árvores AVL com o número mínimo de nodos: um com altura $h - 1$ e o outro com altura $h - 2$. Levando a raiz em consideração, obtemos a seguinte fórmula que relaciona $n(h)$ com $n(h - 1)$ e $n(h - 2)$, para $h \geq 3$:

$$n(h) = 1 + n(h - 1) + (h - 2). \tag{3.1}$$

A Fórmula 3.1 implica que $n(h)$ é uma função crescente de h. Desta forma, sabemos que $n(h - 1) > n(h - 2)$. Substituindo $n(h - 1)$ por $n(h - 2)$ na Fórmula 3.1 e descartando o 1, obtemos, para $h \geq 3$,

$$n(h) > 2 \cdot n(h - 2). \tag{3.2}$$

A Fórmula 3.2 indica que $n(h)$ no mínimo dobra a cada vez que h cresce em 2, o que intuitivamente significa que $n(h)$ cresce exponencialmente. Para mostrar este fato de uma maneira formal, aplicamos a Fórmula 3.2 repetidamente, através de um argumento indutivo, para mostrar que:

$$n(h) > 2^i \cdot n(h - 2i) \tag{3.3}$$

para qualquer inteiro i, tal que $h - 2i \geq 1$. Uma vez que já sabemos os valores de $n(1)$ e $n(2)$, tomamos i de maneira que $h - 2i$ seja igual a 1 ou 2. Ou seja, usamos $i = \lceil h/2 \rceil - 1$. Substituindo o valor de i na Fórmula 3.3, obtemos, para $h \geq 3$,

$$\begin{aligned} n(h) &> 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} n(1) \\ &\geq 2^{\frac{h}{2} - 1}. \end{aligned} \tag{3.4}$$

Tomando os logaritmos de ambos os lados da Fórmula 3.4, temos $\log n(h) > \frac{h}{2} - 1$, a partir do qual obtemos

$$h < 2 \log n(h) + 2, \tag{3.5}$$

que implica que uma árvore AVL armazenando n chaves tem altura de no mínimo $2\log n + 2$. ■

Pelo Teorema 3.2 e a análise das árvores de pesquisa binária vista na Seção 3.1.2, a operação findElement, em um dicionário implementado usando-se uma árvore AVL, executa em tempo $O(\log n)$, onde n é o número de itens no dicionário.

3.2.1 Operações de atualização

O último aspecto importante é mostrar como manter a propriedade do balanceamento de uma árvore AVL após uma inserção ou remoção. As operações de inserção e remoção para árvores AVL são similares àquelas para árvores binárias; porém com árvores AVL, precisamos executar computações adicionais.

Inserção

Uma inserção em uma árvore AVL T inicia como em uma operação insertItem descrita na Seção 3.1.4 para uma árvore binária de pesquisa (simples). Lembramos que esta operação sempre insere o novo item no nodo w de T que foi previamente um nodo externo e transforma w em nodo interno com a operação expandExternal, isto é, adiciona dois nodos filhos externos em w. Esta ação pode violar a propriedade de balanceamento da altura; entretanto, para alguns nodos incrementa sua altura em 1. Em particular, o nodo w e possivelmente alguns de seus ancestrais terão sua altura acrescida em 1. Conseqüentemente, vamos descrever como reestruturar T para restaurar sua altura balanceada.

Dada uma árvore de pesquisa binária T, dizemos que um nodo v de T está ***balanceado*** se o valor absoluto da diferença entre as alturas dos filhos de v é no máximo 1, e dizemos que está ***desbalanceado*** em caso contrário. Então, caracterizar uma árvore AVL pela propriedade de altura/balanceamento equivale a dizer que todos seus nodos internos estão balanceados.

Suponha que T satisfaça à propriedade de altura/balanceamento e, por isso, é uma árvore AVL, antes de inserirmos um novo item. Como mencionamos, depois de executar a operação expandExternal(w) em T, as alturas de alguns nodos de T, incluindo w, crescem. Todos esses nodos estão no caminho de T, que parte de w e vai até a raiz de T, e são os únicos nodos de T que podem ter se desbalanceado (veja a Figura 3.12a). Naturalmente, se isso ocorrer, então T não é mais uma árvore AVL; conseqüentemente, necessitamos de um mecanismo para consertar o "desbalanceamento" recém causado.

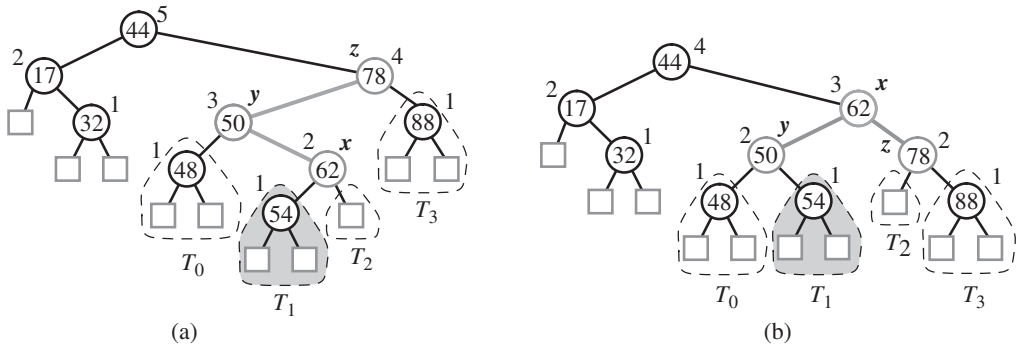


Figura 3.12 Um exemplo de inserção de um elemento com chave 54 na árvore AVL da Figura 3.11: (a) depois da inserção de um novo nodo para a chave 54, os nodos que armazenam as chaves 78 e 44 se tornam desbalanceados; (b) uma reestruturação trinodo restaura a propriedade de altura/balanceamento. Mostramos as alturas dos nodos próximo aos mesmos e identificamos os nodos x, y e z.

Algoritmo restructure(x):

***Entrada*:** um nodo x de uma árvore de pesquisa binária T que tem tanto um pai y como um avô z

***Saída*:** árvore T depois de reestruturação trinodo (que corresponde a uma rotação simples ou dupla) envolvendo os nodos x, y e z.

1: Façamos (a,b,c), da esquerda para direita (interfixado), ser a lista de nodos x, y e z, e façamos (T_0, T_1, T_2, T_3), da esquerda para a direita (interfixado), ser a lista das quatro subárvores de x, y e z não enraizadas em x, y ou z.

2: Substituamos a subárvore enraizada em z por uma nova subárvore enraizada em b.

3: Façamos a ser o filho da esquerda de b e T_0 e T_1 serem as subárvores esquerda e direita de a, respectivamente.

4: Façamos c ser o filho da direita de b e T_2 e T_3 serem as subárvores esquerda e direita de c, respectivamente.

Algoritmo 3.13 Operação de reestruturação trinodo em uma árvore de pesquisa binária.

Restauramos o balanceamento dos nodos em uma árvore binária T através de uma estratégia simples de "pesquise-e-conserte". Em especial, façamos z ser o primeiro nodo que encontramos indo para cima a partir de w em direção à raiz de T, de maneira que z está desbalanceado (veja a Figura 3.12a). Além disso, façamos y indicar os filhos de z com uma altura maior (e observe que y tem de ser um ancestral de w). Finalmente, façamos x ser o filho de y com uma altura maior (e se houver um nó, escolha x para ser o ancestral de w). Observamos que o nodo x, por ser igual a w e x, é o neto de z. Uma vez que z se tornou desbalanceado por uma inserção na subárvore enraizada em seu filho y, a altura de y é duas unidades maior que de seu irmão. Agora rebalancearemos a subárvore enraizada em z chamando o método de ***reestruturação de trinodo***, restructure(x), descrito no Algoritmo 3.13 e ilustrado nas Figuras 3.12 e 3.14. Uma reestruturação trinodo troca temporariamente os nomes dos nodos x, y e z para a, b e c, de maneira que a precede b e b precede c em um caminhamento interfixado sobre T. Existem quatro possibilidades de mapeamento de x, y e z para a, b e c, como pode ser visto na Figura 3.14, os quais são unificados em um caso por nossa troca de rótulos. A reestruturação trinodo substitui z por um nodo chamado b, faz os filhos deste nodo serem a e c e faz os filhos de a e c serem os quatro filhos anteriores de x, y e z (outros que não x e y), enquanto mantém os relacionamento interfixados de todos os nodos de T.

A modificação de uma árvore T causada pela operação de reestruturação trinodo é em geral chamada de ***rotação*** por causa da forma geométrica pela qual podemos visualizar a forma como T se altera. Se $b = y$ (veja o Algoritmo 3.13), o método de reestruturação trinodo é chamado de ***rotação simples***, na medida em que pode ser visualizado como a rotação de y sobre z (veja as Figuras 3.14a e b). Por outro lado, se $b = x$, a operação de reestruturação trinodo é chamada de ***rotação dupla***, na medida que pode ser visualizada como uma rotação inicial de x sobre y e então sobre z (veja a Figura 3.14c e d, e a Figura 3.12). Alguns pesquisadores em computação tratam estes dois tipos de rotação como métodos separados, cada um com dois tipos simétricos; escolhemos, entretanto, unificar os quatro tipos de rotação em uma única operação de reestruturação trinodo. Não importando a forma como é vista, entretanto, observamos que o método de reestruturação trinodo modifica os relacionamentos pai-filho de $O(1)$ nodos de T, enquanto preserva a ordenação do caminhamento interfixado de todos os nodos de T.

Além da propriedade de preservação da ordem, a reestruturação trinodo altera as alturas dos nodos de T para restaurar o balanceamento. Lembramos que executamos o

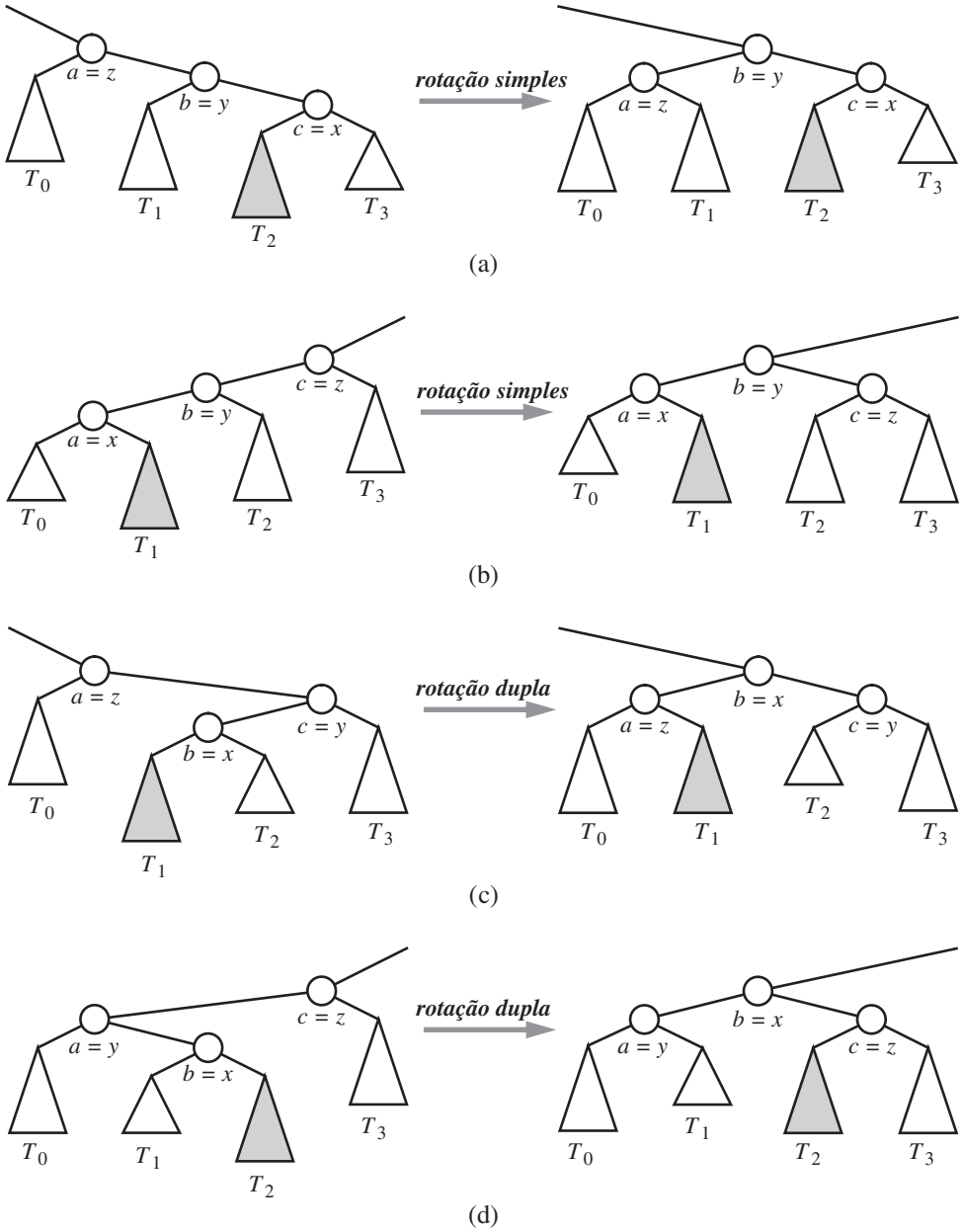


Figura 3.14 Ilustração esquemática de uma operação de reestruturação trinodo (Trecho de Código 3.14). As partes (a) e (b) mostram uma rotação simples, enquanto que as partes (c) e (d) mostram uma rotação dupla.

método restructure(x) porque z, o avô de x, estava desbalanceado. Além disso, este desbalanceamento é conseqüência de um dos filhos de x ter altura muito grande em relação à altura dos outros filhos de z. Como resultado da rotação, movemos para cima o filho "alto" de x, enquanto movemos para baixo o filho "baixo" de z. Desta forma, depois de executarmos restructure(x), todos os nodos na subárvore agora enraizada no

nodo que chamamos de *b* estão balanceados (veja a Figura 3.14). Em conseqüência, restauramos a propriedade de altura/balanceamento ***localmente*** nos nodos *x, y* e *z*.

Além disso, depois de executada uma nova inserção de item, a subárvore enraizada em *b* substitui a que primeiro estava enraizada em *z,* que era mais alta em uma unidade, e todos os ancestrais de *z* que estavam desbalanceados tornam-se balanceados (veja a Figura 3.12) (a justificativa deste fato é deixada para o Exercício C-3.13). Desta forma, essa reestruturação também restaura a propriedade de altura/balanceamento ***globalmente***. Isto é, uma rotação (simples ou dupla) é suficiente para restaurar o balanceamento da altura em uma árvore AVL após uma inserção.

Remoção

Assim como no caso da operação de dicionário insertItem, iniciamos a implementação da operação de dicionário removeElement sobre uma árvore AVL usando o algoritmo para executar esta operação sobre uma árvore binária comum. A dificuldade adicional em usar esta abordagem com uma árvore AVL é que ela pode violar a propriedade de altura/balanceamento.

Em especial, depois de removermos um nodo interno com a operação removeAboveExternal e elevarmos um de seus filhos em seu lugar, pode haver um nodo desbalanceado em *T* no caminho partindo do pai *w* do nodo recém removido até a raiz de *T* (veja a Figura 3.15a). De fato, pode haver no máximo um nodo desbalanceado (a justificativa deste fato fica para o Exercício C-3.12).

Como na inserção, usaremos a reestruturação trinodo para restaurar o balanceamento da árvore *T*. Em especial, faremos *z* ser o primeiro nodo desbalanceado encontrado subindo a partir de *w* em direção à raiz de *T*. Também faremos *y* ser o filho de *z* com maior altura (observamos que o nodo *y* é o filho de *z* que não é ancestral de *w*) e faremos *x* ser o filho de *y* da seguinte maneira: se um dos filhos de *y* for mais alto que outro, faremos *x* ser o filho mais alto de *y*; se ambos os filhos de *y* tiverem a mesma altura, faremos *x* ser o filho de *y* do mesmo lado de *y*, isto é, se *y* for um filho da esquerda, faremos *x* ser um filho da esquerda de *y*. Em qualquer caso, executamos a operação restructure(*x*) que restaura a propriedade de altura/balanceamento ***localmente*** na subárvore que estava primeiramente enraizada em *z* e agora está enraizada no nodo temporariamente chamado de *b* (veja a Figura 3.15b).

Infelizmente, essa reestruturação trinodo pode reduzir a altura da árvore enraizada em *b* em 1, o que pode fazer com que o ancestral de *b* fique desbalanceado. Assim, uma reestruturação trinodo simples não restaura necessariamente a propriedade de altura/balanceamento globalmente depois de uma remoção. Então, depois de rebalan-

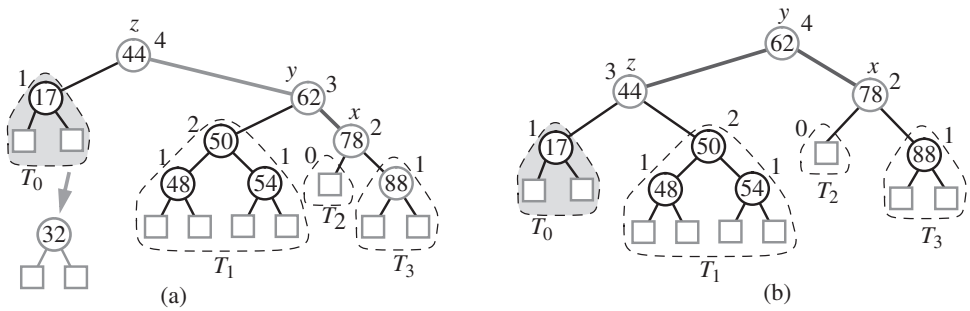


Figura 3.15 Remoção do elemento com chave 32 da árvore AVL da Figura 3.11: (a) depois da remoção do nodo que armazena a chave 32, a raiz se torna desbalanceada; (b) uma rotação (simples) restaura a propriedade de altura/balanceamento.

cear z, continuamos caminhando para cima em T, procurando por nodos desbalanceados. Se encontrarmos outro, executamos a operação de reestruturação para restaurar seu balanceamento e continuamos avançando para cima em T, procurando por mais, sempre em direção à raiz. Ainda, uma vez que a altura de T é $O(\log n)$, onde n é o número de itens, pelo Teorema 3.2, $O(\log n)$ reestruturações trinodo são suficientes para restaurar a propriedade de altura/balanceamento.

3.2.2 Desempenho

Resumimos a análise de uma árvore AVL como segue. As operações findElement, insertItem e removeElement visitam os nodos ao longo de um caminho raiz-folha de T, além de, possivelmente, seus irmãos e gasta um tempo $O(1)$ por nodo. Desta forma, uma vez que a altura de T seja $O(\log n)$ pelo Teorema 3.2, cada uma das operações anteriores leva um tempo $O(\log n)$. Demonstramos este desempenho na Figura 3.16.

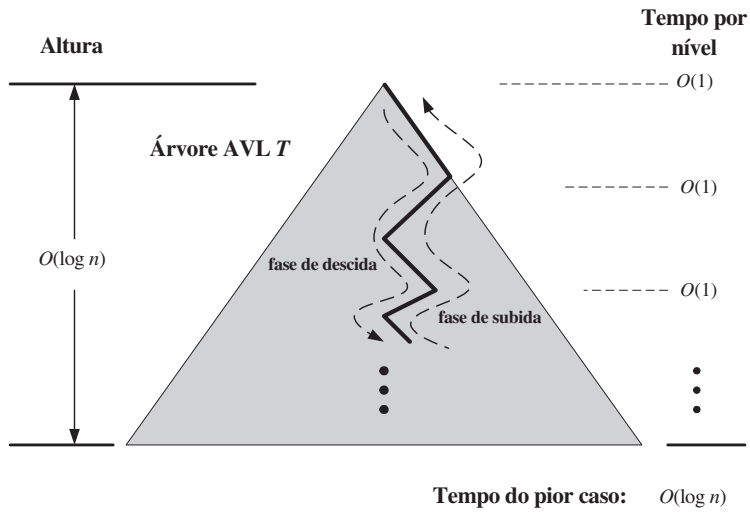


Figura 3.16 Ilustração do tempo de execução de pesquisas e atualizações em uma árvore AVL. O desempenho de tempo é $O(1)$ por nível, dividido em uma fase de descida, que tipicamente envolve pesquisa, e uma fase de subida, que tipicamente envolve atualização dos valores das alturas e execução de restruturações trinodo locais (rotações).

3.3 Árvores de pesquisa com profundidade limitada

Algumas estruturas de pesquisa baseiam sua eficiência em regras que explicitamente limitam sua profundidade. Tipicamente essas árvores definem uma função de profundidade ou uma "pseudofunção de profundidade" intimamente relacionada com a profundidade, de maneira que todos os nodos externos se encontrem à mesma profundidade ou pseudoprofundidade. Desta forma, elas mantêm todos os nodos externos a uma profundidade $O(\log n)$ em uma árvore que armazena n elementos. Como as pesquisas e atualizações normalmente executam em tempos que são proporcionais à profundidade, ***árvores com profundidade limitada*** podem ser usadas para implementar dicionários ordenados com tempos de pesquisa e atualização $O(\log n)$.

3.3.1 Árvores de pesquisa genéricas

Algumas árvores de pesquisa com profundidade limitada são árvores genéricas, isto é, árvores com nodos internos que podem ter dois ou mais filhos. Nesta seção, descrevemos como árvores genéricas podem ser usadas como árvores de pesquisa, incluindo a forma como árvores genéricas armazenam itens e como podemos executar operações de pesquisa em árvores de pesquisa genéricas. Lembramos que os ***itens*** que armazenamos em uma árvore de pesquisa são pares da forma (k, x) onde k é a ***chave*** e x é o elemento associado com a chave.

Seja v um nodo de uma árvore ordenada. Dizemos que v é um ***d-nodo*** se v tem d filhos. Definimos uma ***árvore genérica de pesquisa*** como sendo uma árvore ordenada T que tem as seguintes propriedades (que são ilustradas na Figura 3.17a):

- Cada nodo interno de T tem ao menos dois filhos. Isto é, cada nodo interno é um d-nodo, onde $d \geq 2$.
- Cada nodo interno de T armazena uma coleção de itens da forma (k,x), onde k é a chave e x é o elemento.
- Cada d-nodo v de T, com filhos $v_1, ..., v_d$ armazena $d - 1$ itens $(k_1,x_1), ..., (k_{d-1}, x_{d-1})$, onde $k_1 \leq ... \leq k_{d-1}$.
- Definiremos, por convenção, $k_0 = -\infty$ e $k_d = +\infty$. Para cada item (k,x) armazenado em um nodo da subárvore de v enraizada em v_i, $i = 1, ..., d$, temos que $k_{i-1} \leq k \leq k_i$.

Ou seja, se considerarmos o conjunto de chaves armazenadas em v, incluindo as chaves fictícias especiais $k_0 = -\infty$ e $k_d = +\infty$, então uma chave k armazenada na subárvore de T enraizada no nodo filho v_i deve estar "entre" duas chaves armazenadas em v. Este ponto de vista simples origina a regra que diz que um nodo com d filhos armazena $d - 1$ chaves regulares e forma a base do algoritmo de pesquisa em uma árvore genérica de pesquisa.

Pela definição acima, os nodos externos de uma árvore genérica de pesquisa não armazenam nenhum item, servindo apenas como "guardadores de locais". Desta forma, vemos uma árvore binária de pesquisa (Seção 3.1.2.) como um caso especial de árvore genérica de pesquisa. No extremo oposto, uma árvore genérica de pesquisa pode ter apenas um único nodo interno que armazena todos os itens. Além disso, apesar de os nodos externos poderem ser null, partimos da hipótese simplificada de que são nodos que não armazenam nada.

Tendo os nodos internos de uma árvore genérica dois ou mais filhos, entretanto, existe uma relação interessante entre o número de itens e o número de nodos externos.

Teorema 3.3: *uma árvore de pesquisa genérica que armazena n itens tem n + 1 nodos externos.*

Deixamos a justificativa dessa proposição como exercício (C-3.14).

Pesquisa em uma árvore genérica

Dada uma árvore genérica T, pesquisar por um elemento com chave k é simples. Executamos tal pesquisa seguindo um caminho em T que se inicia na raiz (veja a Figura 3.17b e c). Quando estivermos em um d-nodo v durante esta pesquisa, vamos comparar a chave k com as chaves $k_1, ... , k_{d-1}$ armazenadas em v. Se $k = k_i$ para algum i, a pes-

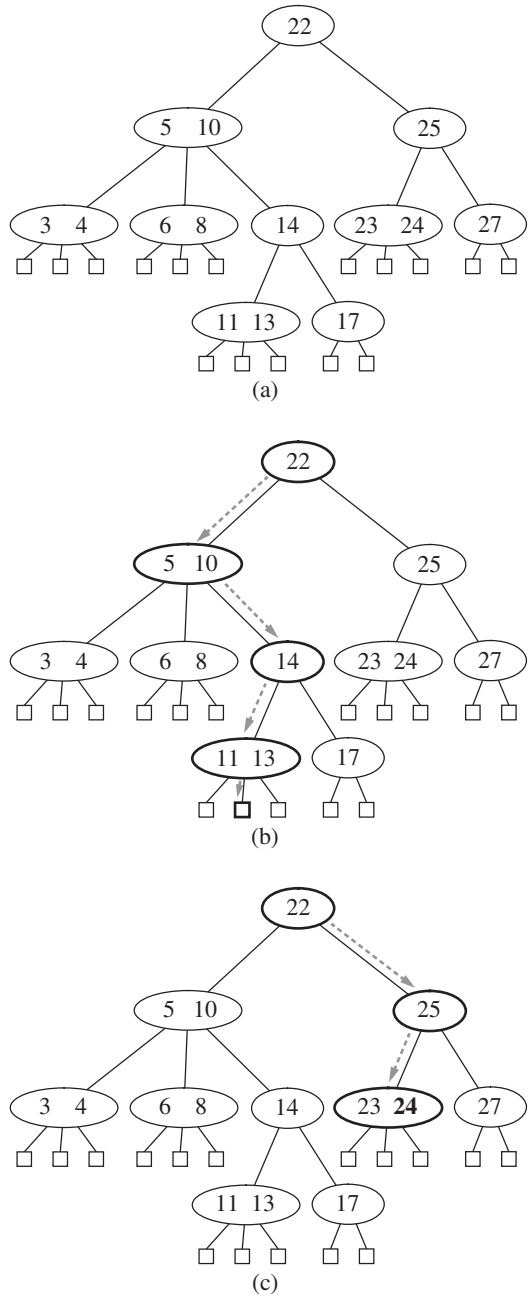


Figura 3.17 (a) Árvore genérica de pesquisa T; (b) caminho pesquisado em T para a chave 12 (pesquisa sem sucesso); (c) caminho pesquisado em T para a chave 24 (pesquisa com sucesso).

quisa é encerrada com sucesso. Caso contrário, continuamos a pesquisa no filho v_i de v de maneira que $k_{i-1} < k < k_i$. (Lembre que consideramos $k_0 = -\infty$ e $k_d = +\infty$.) Se atingimos um nodo externo, então sabemos que não há nenhum item com a chave k em T, e a pesquisa termina sem sucesso.

Estruturas de dados para árvores de pesquisa genéricas

Na Seção 2.3.4, discutimos diferentes maneiras de representar árvores genéricas. Cada uma destas representações também pode ser reutilizada para árvores de pesquisa genéricas. Na verdade, ao usarmos uma árvore genérica para implementar uma árvore de pesquisa genérica, a única informação adicional que precisamos armazenar em cada nodo é o conjunto de itens (incluindo as chaves) associados com os mesmos. Ou seja, precisamos armazenar em v uma referência para um contêiner ou objeto coleção que armazene os itens de v.

Lembremos que, quando usamos uma árvore binária para representar um dicionário ordenado D, simplesmente armazenamos uma referência para um único item em cada nodo interno. Usando uma árvore de pesquisa genérica T para representar D, devemos armazenar uma referência para um conjunto ordenado de itens associados com v em cada nodo interno v de T. Essa argumentação pode parecer recursiva em um primeiro momento, uma vez que necessitamos de uma representação de um dicionário ordenado para representar um dicionário ordenado. Podemos evitar essa recursividade, entretanto, usando a técnica ***bootstrapping***, em que usamos a solução anterior (menos desenvolvida) de um problema para uma criar uma solução nova (mais avançada). Neste caso, o *bootstrapping* consiste em representar o conjunto ordenado associado com cada nodo interno usando a estrutura de dados para dicionário que construímos anteriormente (por exemplo, uma tabela de pesquisa baseada em um vetor ordenado, como mostrado na Seção 3.1.1). Em particular, supondo que dispomos de uma maneira de implementar dicionários ordenados, podemos implementar uma árvore de pesquisa genérica usando uma árvore T e armazenando tal dicionário em cada d-nodo v de T.

O dicionário que armazenamos em cada nodo v é conhecido como uma estrutura de dados ***secundária***, na medida em que é usada para suportar a estrutura de dados maior, a ***primária***. Indicamos o dicionário armazenado no nodo v de T como $D(v)$. Os itens que armazenamos em $D(v)$ nos permitem determinar para qual nodo filho devemos ir durante uma operação de pesquisa. Especificamente, para cada nodo v de T, com filhos $v_1, \ldots, v_d$ e itens $(k_1, x_1), \ldots, (k_{d-1}, x_{d-1})$, armazenamos no dicionário $D(v)$ os itens $(k_1, x_1, v_1), (k_2, x_2, v_2), \ldots, (k_{d-1}, x_{d-1}, v_{d-1}), (+\infty, null, v_d)$. Ou seja, um item (k_i, x_i, v_i) de um dicionário $D(v)$ tem chave k_i e elemento (x_i, v_i). Observamos que o último ítem armazena a chave especial $+\infty$.

Com esta implementação de uma árvore de pesquisa genérica T, o processamento de um d-nodo v durante uma pesquisa por um elemento de T com chave k pode ser feita executando-se uma operação de pesquisa para encontrar o item (k_i, x_i, v_i) em $D(v)$ com a menor chave maior ou igual a k, da mesma forma que na operação closestElemAfter(k) (veja a Seção 3.1). Distinguimos dois casos:

- Se $k < k_i$, então continuamos a pesquisa processando o filho v_i .(Observamos que, se a chave especial $k_d = +\infty$ é retornada, então k é maior do que todas as chaves armazenadas no nodo v, e continuamos a pesquisa processando o filho v_d.)
- Por outro lado, se $(k = k_i)$, então a pesquisa se encerra com sucesso.

Questões de desempenho de árvores de pesquisa genéricas

Considere o espaço necessário para essa implementação de uma árvore de pesquisa genérica T que armazena n itens. Pelo Teorema 3.3, usando qualquer uma das implementações normais de dicionários ordenados (Seção 2.5) para as estruturas secundárias dos nodos de T, o espaço total requisitado para T é $O(n)$.

Consideremos agora o tempo gasto respondendo a uma pesquisa sobre T. O tempo gasto em um d-nodo v de T durante a pesquisa depende de como implementamos a estrutura secundária $D(v)$. Se $D(v)$ é implementado usando uma seqüência baseada em um vetor ordenado (isto é, uma tabela de pesquisa), então podemos processar v em tempo $O(\log d)$. Se, por outro lado, $D(v)$ é implementado usando uma seqüência não- ordenada (isto é, um arquivo de *log*), então o processamento de v leva tempo $O(d)$. Façamos $d_{\max}$ denotar o número máximo de filhos de qualquer nodo de T, e h denotar a altura de T. O tempo de pesquisa em uma árvore de pesquisa genérica será tanto $O(h\, d_{\max})$ ou $O(h \log d_{\max})$, dependendo da implementação específica das estruturas secundárias nos nodos de T (os dicionários $D(v)$). Se $d_{\max}$ for uma constante, o tempo de execução para efetivar uma pesquisa é $O(h)$, independentemente da implementação da estrutura secundária.

Desta forma, o objetivo principal visando à eficiência em uma árvore de pesquisa genérica é manter a altura o menor possível, ou seja, queremos que h seja uma função logarítmica de n, o número total de itens armazenados no dicionário. Uma árvore de pesquisa com altura logarítmica como esta é chamada de ***árvore de pesquisa balanceada***. Árvores de pesquisa com profundidade limitada satisfazem esse objetivo mantendo os nodos externos exatamente no mesmo nível de profundidade.

Discutiremos na seqüência uma árvore de pesquisa com profundidade limitada que corresponde a uma árvore de pesquisa genérica que fixa $d_{\max}$ em 4. Na Seção 14.1.2, discutiremos outros tipos de árvores de pesquisa genéricas que tenham aplicações onde nossa árvore de pesquisa é muito grande para poder ser armazenada em sua totalidade na memória principal do computador.

3.3.2 Árvores (2,4)

Quando se usam árvores de pesquisa genéricas na prática, é desejável que a mesma seja balanceada, isto é, tenha altura logarítmica. A árvore de pesquisa genérica que estudaremos a seguir é fácil de manter balanceada – é a árvore (2,4), também chamada de árvore 2-4 ou árvore 2-3-4. De fato, podemos manter o balanço em uma árvore (2,4) mantendo duas propriedades simples (veja a Figura 3.18):

Propriedade do tamanho: cada nodo tem no máximo quatro filhos.

Propriedade da profundidade: todos nodos externos têm a mesma profundidade.

Garantir a propriedade do tamanho para árvores (2,4) mantém o tamanho dos nodos de uma árvore genérica de pesquisa constante, permitindo-nos representar o dicionário $D(v)$ armazenado em cada nodo interno v usando um arranjo de tamanho fixo. A propriedade da profundidade, por outro lado, reforça o balanço de uma árvore (2,4), forçando-a a ser uma estrutura de profundidade limitada.

Teorema 3.4: *a altura de uma árvore* (2,4) *que armazena n itens é* $\Theta(\log n)$.

Prova: seja h a altura de uma árvore (2,4) T que armazena n itens. Observamos que, pela propriedade do tamanho, temos no máximo quatro nodos na profundidade 1, no máximo 4^2 na profundidade 2, e assim por diante. Assim, o número de nodos externos em T é no máximo 4^h. Da mesma forma, pela propriedade da profundidade e pela definição de uma árvore (2,4), devemos ter no mínimo dois nodos na profundidade 1, pelo menos 2^2 na profundidade 2, e assim por diante. Assim, o número de nodos externos em T é no mínimo 2^h. Além disso, pelo Teorema 3.3, o número de nodos externos em T é $n + 1$. Por essa razão, obtemos

$$2^h \leq n + 1 \quad \text{e} \quad n + 1 \leq 4^h.$$

Considerando o logaritmo de base 2 para cada um dos termos anteriores, obtemos que

$$h \leq \log(n + 1) \quad \text{e} \quad \log(n + 1) \leq 2h.$$

o que justifica nosso teorema.

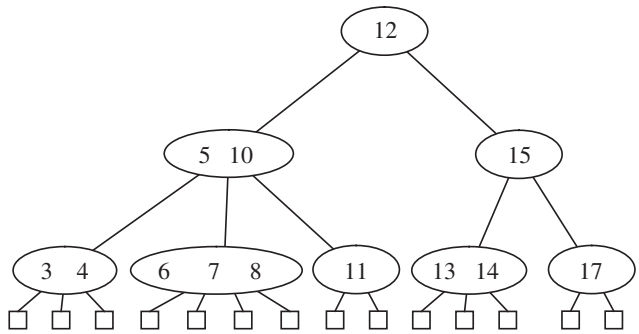


Figura 3.18 Uma árvore (2,4).

Inserção em uma árvore (2,4)

O Teorema 3.4 afirma que as propriedades do tamanho e profundidade são suficientes para manter uma árvore genérica balanceada. Entretanto, manter essas propriedades após executar operações de inserção e remoção em uma árvore (2,4) requer algum esforço.

Em particular, para inserir um novo item (k, x), com chave k, em uma árvore (2,4) T, primeiramente pesquisamos k. Supondo que T não contenha nenhum elemento com a chave k, esta pesquisa irá terminar sem sucesso em um nodo externo z. Seja v o nodo pai de z. Inserimos o novo item no nodo v e acrescentamos um novo filho w (um nodo externo) em v, à esquerda de z. Ou seja, adicionamos o item (k,x,w) ao dicionário $D(v)$.

Nosso método de inserção preserva a propriedade da profundidade, uma vez que adicionamos um novo nodo externo no mesmo nível dos nodos externos existentes. Por outro lado, ela pode violar a propriedade do tamanho. Na verdade, se um nodo v era um 4-nodo, então ele pode se tornar um 5-nodo após a inserção, o que faria com que a árvore T não fosse mais uma árvore (2,4). Esse tipo de violação da propriedade do tamanho é chamado de ***overflow*** do nodo v e deve ser resolvido de maneira a restaurar as propriedades da árvore (2,4). Sejam $v_1, \ldots, v_5$ os filhos de v e sejam $k_1, \ldots, k_4$ as chaves armazenadas em v. Para remediar o *overflow* do nodo v, executaremos uma operação de ***divisão*** em v como segue (veja a Figura 3.19):

- Substitua v por dois nodos v' e v'', onde
  - v' é um 3-nodo com filhos v_1, v_2, v_3 que armazenam as chaves k_1 e k_2;
  - v'' é um 2-nodo com filhos v_4, v_5 que armazenam a chave k_4.
- Se v era a raiz de T, crie um novo nodo raiz u; senão, faça u ser o pai de v.
- Insira a chave k_3 em u e faça v' e v'' filhos de u, de maneira que, se v fosse o i-ésimo filho de u, então v' e v'' passam a ser os filhos i e $i + 1$ de u, respectivamente.

Observamos uma seqüência de inserções em uma árvore (2,4) na Figura 3.20.

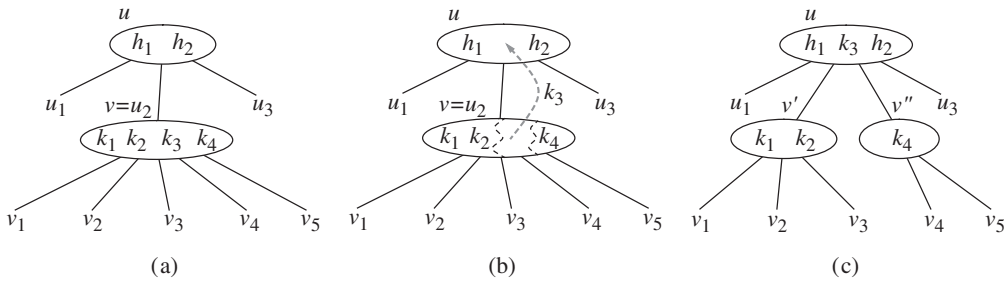


Figura 3.19 Divisão de um nodo: (a) *overflow* no 5-nodo v; (b) terceira chave de v inserida no pai u de v; (c) nodo v substituído por um 3-nodo v' e um 2-nodo v''.

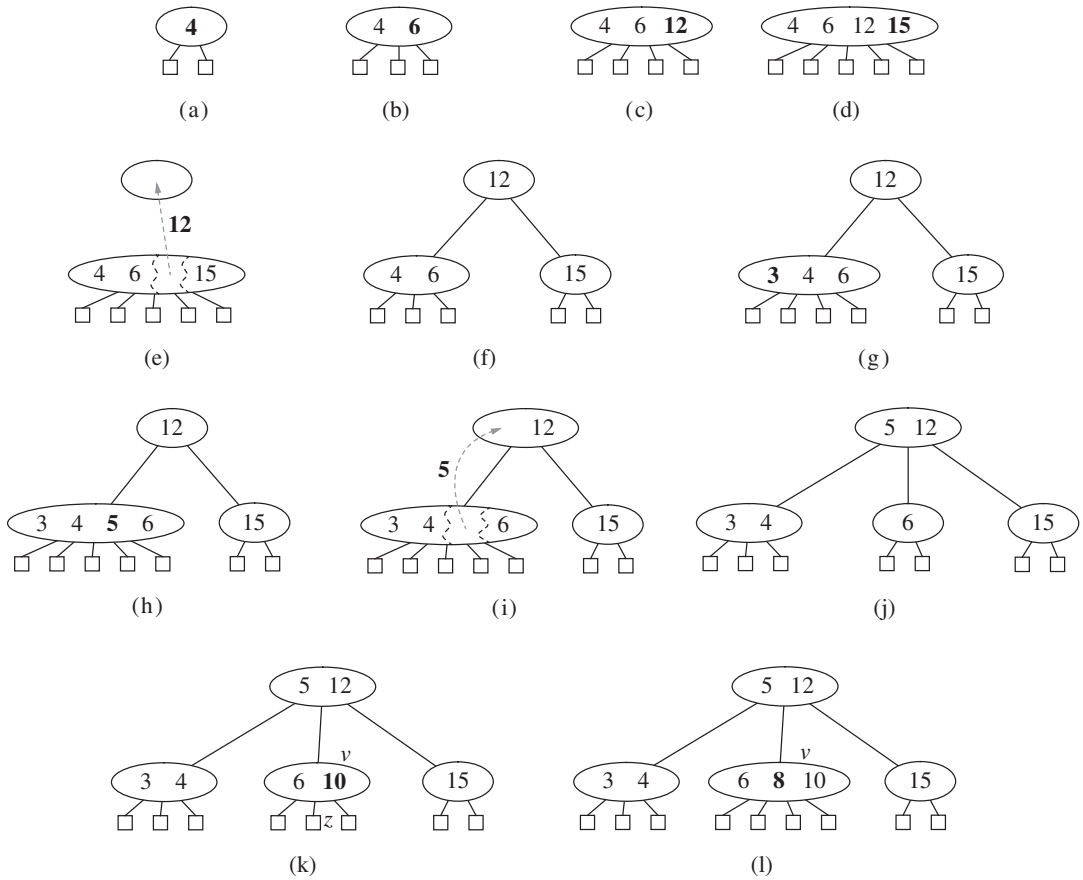


Figura 3.20 Seqüência de inserções em uma árvore (2,4): (a) árvore inicial com um item; (b) inserção de 6; (c) inserção de 12; (d) inserção de 15, causando um *overflow*; (e) divisão, implica a criação de um novo nodo raiz; (f) após a divisão; (g) inserção de 3; (h) inserção de 5, causando um *overflow*; (i) divisão; (j) após a divisão; (k) inserção de 10; (l) inserção de 8.

Analisando inserções em uma árvore (2,4)

Como conseqüência de uma operação de divisão sobre um nodo *v,* um novo *overflow* pode ocorrer no pai *u* de *v*. Se esse *overflow* ocorrer, ele dispara, por sua vez, uma divisão no nodo *u* (veja a Figura 3.21). Uma operação de divisão pode tanto eliminar como propagar um *overflow* para o pai do nodo corrente. Assim, esta propagação pode continuar até a raiz da árvore de pesquisa. Mas se não propagar-se até a raiz, será finalmente resolvida naquele ponto. Mostraremos uma seqüência de divisões que se propagam na Figura 3.21.

Desta forma, o número de operações de divisão é limitado pela altura da árvore, que corresponde a $O(\log n)$ pelo Teorema 3.4. Sendo assim, o tempo total para executar uma inserção em uma árvore (2,4) é $O(\log n)$.

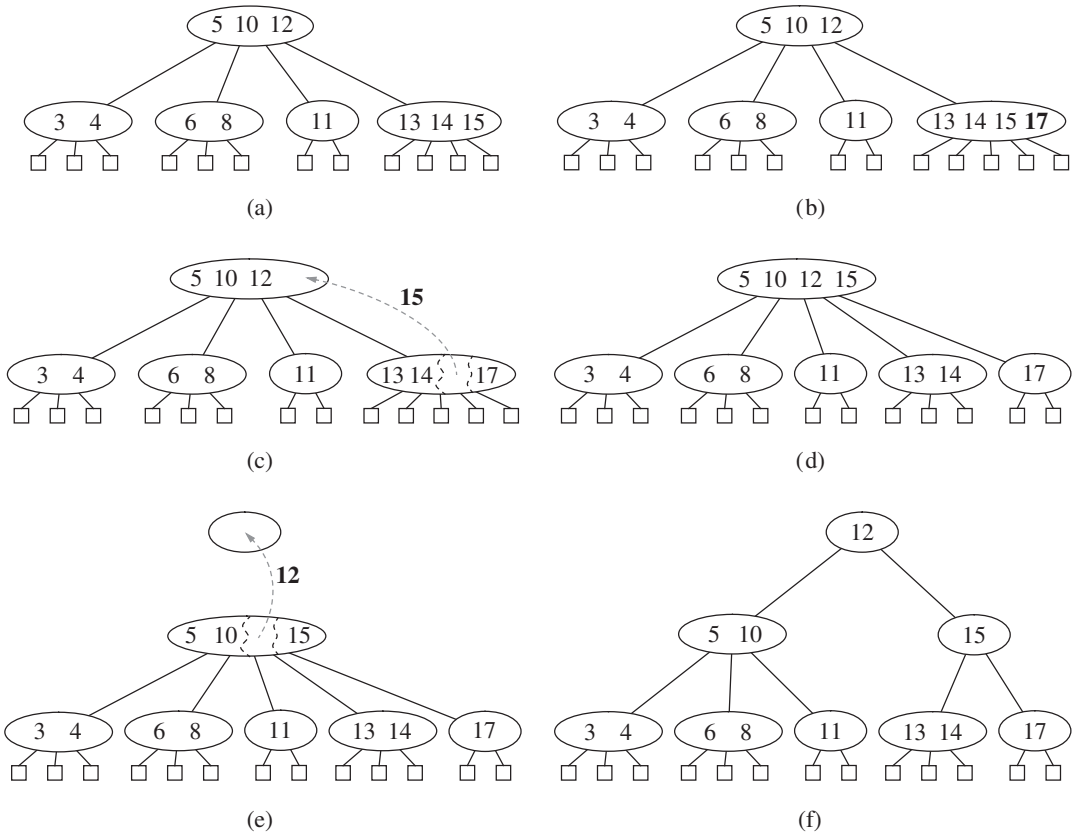


Figura 3.21 Uma inserção em uma árvore (2,4) causa divisões em cascata: (a) antes da inserção; (b) inserção de 17 causando *overflow*; (c) uma divisão; (d) após a divisão, um novo *overflow* ocorre; (e) outra divisão criando um novo nodo raiz; (f) árvore final.

Remoção em uma árvore (2,4)

Consideremos agora a remoção de um item com chave *k* de uma árvore (2,4) *T*. Iniciamos essa operação executando uma pesquisa em *T* por um item com chave *k*. A remoção deste item de uma árvore (2,4) sempre pode cair no caso em que o item a ser removido está armazenado em um nodo *v* cujos filhos são nodos externos. Suponhamos, por

exemplo, que o item com chave k que desejamos remover está armazenado no i-ésimo item (k_i, x_i) no nodo z, que tem apenas nodos internos como filhos. Neste caso, trocamos o item (k_i, x_i) por um item apropriado que esteja armazenado no nodo v com nodos externos como filhos, como segue (Figura 3.22d):

1. Encontramos o nodo interno v mais à direita da subárvore enraizada no i-ésimo filho de z, notando que os filhos do nodo v são todos nodos externos.
2. Trocamos o item (k_i, x_i) de z pelo último item de v.

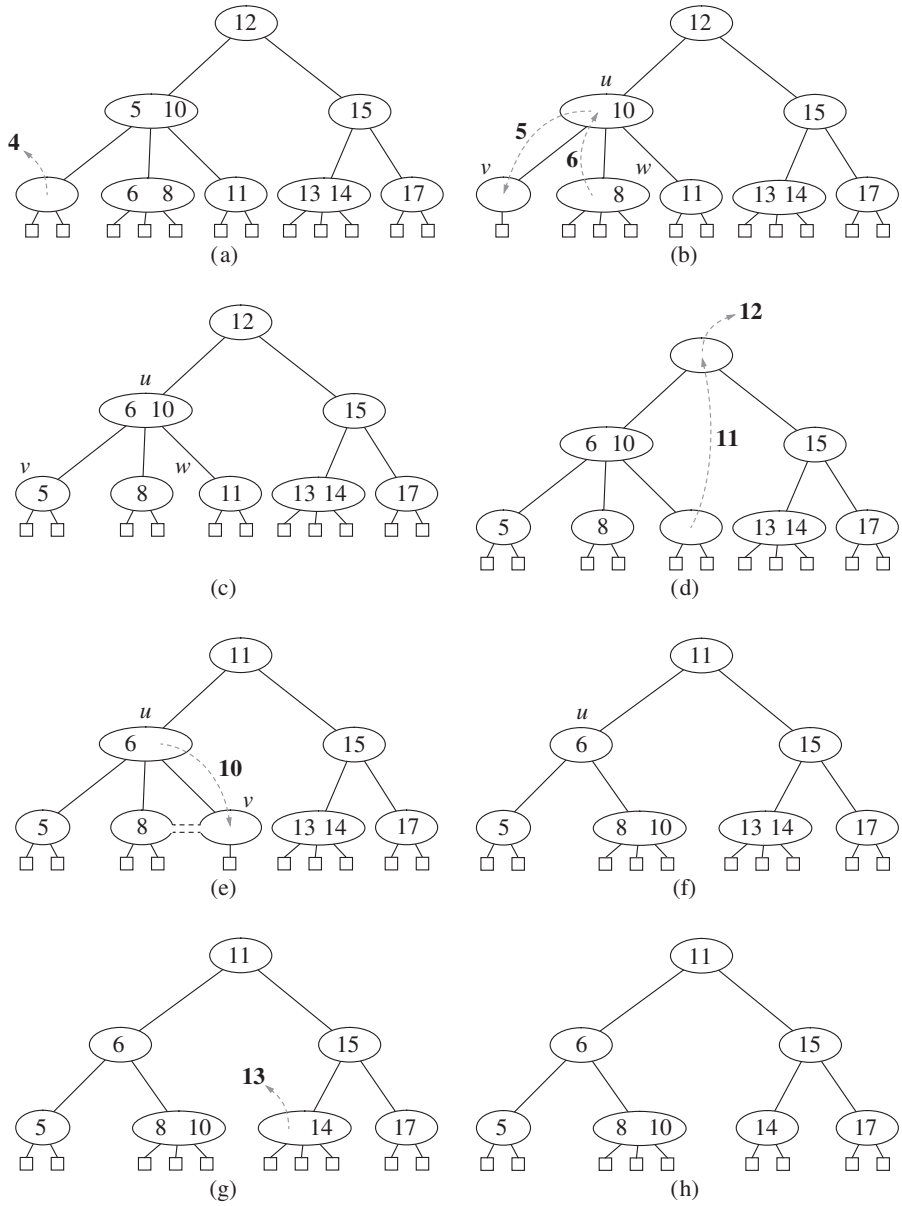


Figura 3.22 Seqüência de remoções de uma árvore (2,4): (a) remoção de 4, causando *underflow*; (b) operação de transferência; (c) após a operação de transferência; (d) remoção de 12, causando *underflow*; (e) operação de fusão; (f) após a operação de fusão; (g) remoção de 13; (h) após a remoção de 13.

Uma vez que garantimos que o item a ser removido está armazenado em um nodo v que tem apenas nodos externos como filhos (porque já estava em v ou porque o tiramos de v), simplesmente removemos o item de v (isto é, do dicionário $D(v)$) e o i-ésimo nodo externo de v.

A remoção de um item (e um filho) de um nodo v, como descrito anteriormente, preserva a propriedade da profundidade, porque sempre removemos um nodo externo filho de um nodo v que tenha apenas nodos externos como filhos. Entretanto, removendo nodos externos, podemos violar a propriedade do tamanho em v. Na verdade, se v era um 2-nodo, então ele se torna um 1-nodo sem itens após a remoção (Figuras 3.22d e e), o que não é permitido em uma árvore (2,4). Esse tipo de violação da propriedade do tamanho é chamado de ***underflow*** do nodo v. Para remediar um *underflow*, verificamos quando um irmão de v é um 3-nodo ou um 4-nodo. Se encontramos tal irmão w, então executamos uma operação de ***transferência***, na qual movemos um filho de w para v, uma chave de w para o pai u de v e uma chave de u para v (veja a Figura 3.22b e c). Se v tiver apenas um irmão, ou se os dois irmãos imediatos de v forem 2-nodos, então executamos uma operação de ***fusão***, na qual unimos v com um irmão, criando um novo nodo v', e movemos uma chave do pai u de v para v' (veja a Figura 3.23e e f).

Uma operação de fusão no nodo v pode causar um novo *underflow* que irá ocorrer no pai u de v, que por sua vez dispara uma transferência ou fusão em u (veja a Figura 3.23). Então, o número de operações de fusão é limitado pela altura da árvore, que é $O(\log n)$ pelo Teorema 3.4. Se um *underflow* se propaga até a raiz, então esta é simplesmente removida (veja a Figura 3.23c e d). Apresentamos uma seqüência de remoções de uma árvore (2,4) nas Figuras 3.22 e 3.23.

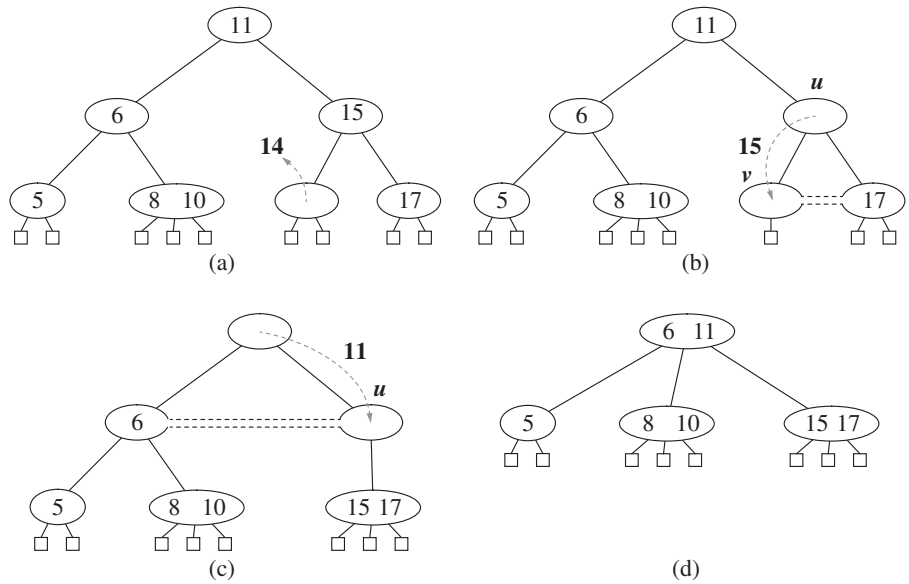


Figura 3.23 Propagação de uma seqüência de fusões em uma árvore (2,4): (a) remoção de 14, causando um *underflow*; (b) fusão, causando outro *underflow*; (c) segunda operação de fusão, causando a remoção da raiz; (d) árvore final.

Desempenho de uma árvore (2,4)

A Tabela 3.24 resume os tempos de execução das principais operações de um dicionário implementado usando uma árvore (2,4). A análise de complexidade do tempo é baseada no seguinte:

- A altura de uma árvore (2,4) que armazena n itens é $O(\log n)$, pelo Teorema 3.4.
- Uma operação de divisão, transferência ou fusão leva tempo $O(1)$.
- Uma pesquisa, inserção ou remoção de um item visita $O(\log n)$ nodos.

Operação	Tempo
size, isEmpty	$O(1)$
findElement, insertItem, removeElement	$O(\log n)$
findAllElements, removeAllElements	$O(\log n + s)$

Tabela 3.24 Desempenho de um dicionário com n elementos implementado usando uma árvore (2,4), onde s denota o tamanho dos iteradores retornados por findAllElements e removeAllElements. O espaço utilizado é $O(n)$.

Desta forma, árvores (2,4) oferecem operações rápidas de pesquisa e alteração em dicionários. Árvores (2,4) também têm um relacionamento interessante com a estrutura de dados que discutiremos a seguir.

3.3.3 Árvores vermelho-pretas

A estrutura de dados que discutiremos nesta seção, a árvore vermelho-preta, é uma árvore de pesquisa binária que usa um tipo de "pseudoprofundidade" para alcançar o balanceamento usando a abordagem de uma árvore de pesquisa com profundidade limitada. Em particular, uma **árvore vermelho-preta** é uma árvore de pesquisa binária com nodos coloridos de vermelho e preto de forma a satisfazer as seguintes propriedades:

***Propriedade da raiz*:** a raiz é preta.

***Propriedade externa*:** todo nodo externo é preto.

***Propriedade interna*:** os filhos de um nodo vermelho são pretos.

Propriedade da profundidade*:** todos os nodos externos tem a mesma ***profundidade preta que é definida como o número de ancestrais pretos menos um.

Um exemplo de árvore vermelho-preta é apresentado na Figura 3.25. No decorrer desta seção, usaremos a convenção de desenhar os *nodos pretos* e as arestas de seus pais com *linhas grossas*.

Como tem sido convencionado neste capítulo, assumimos que os itens são armazenados nos nodos internos da árvore vermelho-preta, com os nodos externos sendo lugares vagos. Além disso, descrevemos nossos algoritmos assumindo que são nodos reais, mas notamos que, ao custo de algoritmos de pesquisa e atualização um pouco mais complicados, nodos externos podem ser **null** ou referências para um objeto NULL_NODE.

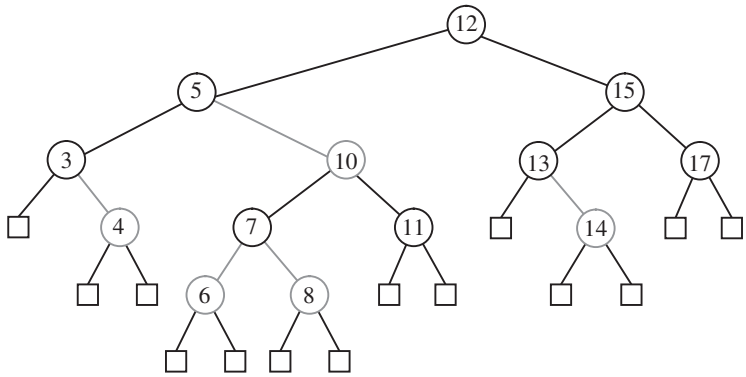


Figura 3.25 Árvore vermelho-preta relacionada com a árvore (2,4) da Figura 3.18. Cada nodo externo desta árvore vermelho-preta tem três ancestrais pretos; portanto, tem profundidade preta 3. Lembre-se de que usamos linhas grossas para denotar os nodos pretos.

A definição de uma árvore vermelho-preta torna-se mais intuitiva, observando-se uma correspondência interessante entre árvores vermelho-pretas e árvores (2,4), como demonstrado na Figura 3.26. Isto é, dada uma árvore vermelho-pretas, podemos construir a árvore (2,4) correspondente, combinando todo nodo vermelho v com seu pai e armazenando o item de v no seu pai. Da mesma forma, podemos transformar qualquer

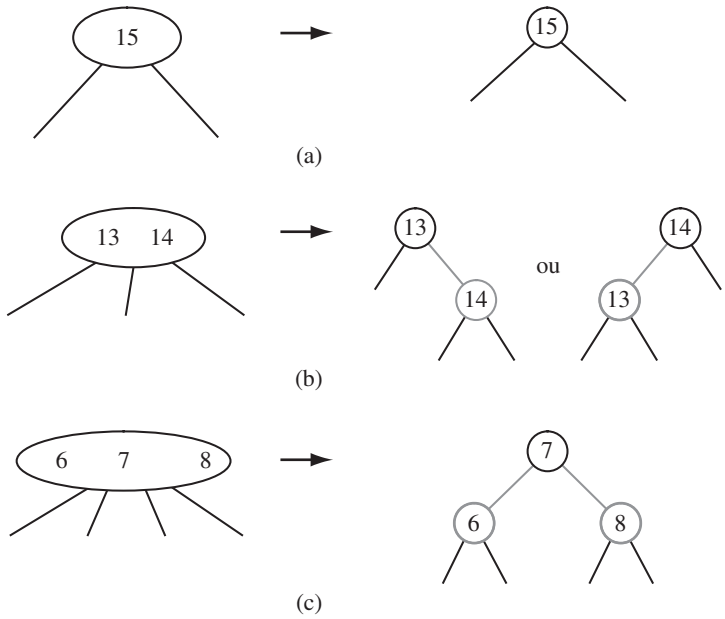


Figura 3.26 Correspondência entre uma árvore (2,4) e uma árvore vermelho-preta: (a) 2-nodo; (b) 3-nodo; (c) 4-nodo.

árvore (2,4) em sua árvore vermelho-preta correspondente, colorindo cada nodo de preto e executando as seguintes transformações sobre cada nodo interno v.

- Se v é um 2-nodo, então mantenha os filhos (pretos) de v como estão.
- Se v é um 3-nodo, então crie um novo nodo vermelho w, passe os primeiros dois filhos (pretos) de v para w e faça w e o terceiro filho de v serem os filhos de v.
- Se v é um 4-nodo, então crie dois novos nodos vermelhos w e z, passe os dois primeiros filhos (pretos) de v para w, passe os dois últimos filhos (pretos) de v para z e faça w e z serem os dois filhos de v.

A correspondência entre árvores (2,4) e árvores vermelho-pretas provê subsídios importantes que serão usados em nossa discussão. Na verdade, os algoritmos de atualização para árvores vermelho-pretas são misteriosos e complexos sem estes subsídios. Consideramos também a seguinte propriedade para árvores vermelho-pretas.

Teorema 3.5: *a altura de uma árvore vermelho-preta que armazena n itens é* $O(\log n)$.

Prova: seja T uma árvore vermelho-preta que armazena n itens e seja h a altura de T. Justificamos esta proposição estabelecendo o seguinte fato:

$$\log(n + 1) \leq h \leq 2 \log(n + 1)$$

Seja d a altura comum a todos os nodos externos de T. Seja T' a árvore (2,4) associada com T e seja h' a altura de T'. Sabemos que $h' = d$. Desta forma, pelo Teorema 3.4, $d = h' \leq \log(n + 1)$. Pela propriedade interna do nodo, $h \leq 2d$. Desta forma, obtemos $h \leq 2 \log(n + 1)$. A outra desigualdade, $\log(n + 1) \leq h$, segue o Teorema 2.8 e o fato de que T tem n nodos internos. ■

Assumimos que uma árvore vermelho-preta é implementada usando a estrutura encadeada de árvores binárias (Seção 2.3.4), na qual armazenamos um item do dicionário e um indicador de cor em cada nodo. Dessa maneira, o espaço necessário para armazenar n chaves é $O(n)$. O algoritmo para pesquisar em uma árvore vermelho-preta T é o mesmo que o para uma árvore de pesquisa binária (Seção 3.1.2). Assim, pesquisar em uma árvore vermelho-preta leva tempo $O(\log n)$.

Executar operações de atualização em uma árvore vermelho-preta é semelhante a executar essas operações em uma árvore de pesquisa binária, exceto pelo fato de que temos de restaurar as propriedades das cores.

Inserção em uma árvore vermelho-preta

Consideremos a inserção de um elemento x com chave k em uma árvore vermelho-preta T, tendo em mente a correspondência entre T e a árvore (2,4) associada T' e o algoritmo de inserção de T'. O algoritmo de inserção inicia como em uma árvore de pesquisa binária (Seção 3.1.4). Isto é, pesquisamos k em T até encontrar um nodo externo de T e substituímos este nodo por um nodo interno z, armazenando (k, x) e obtendo dois nodos filhos externos. Se z é a raiz de T, colorimos z de preto; senão, colorimos z de vermelho. Colorimos também os filhos de z de preto. Esta ação corresponde a inserir (k, x) em um nodo da árvore (2,4) T' com filhos externos. Além disso, essa ação preserva a raiz e as propriedades externa e de profundidade de T, mas pode violar a propriedade interna. Na verdade, se z não é a raiz de T e o pai v de z é vermelho, então te-

mos um pai e um filho (isto é, v e z) que são ambos vermelhos. Observamos que, pela propriedade da raiz, v não pode ser a raiz de T, e pela propriedade interna (que foi anteriormente satisfeita), o pai de u de v tem de ser preto. Uma vez que z e seu pai são vermelhos, mas o avô de z, u, é preto, chamamos esta violação da propriedade interna de ***vermelho duplo*** no nodo z.

Para remediar um vermelho duplo consideramos dois casos.

Caso 1: o irmão *w* de *v* é preto (veja a Figura 3.27). Neste caso, o vermelho duplo indica o fato de que criamos em nossa árvore vermelho-preta uma substituição mal formada para um 4-nodo correspondente da árvore (2,4) T', que tem como seus filhos os quatro filhos pretos de u, v e z. Essa substituição mal formada tem um nodo vermelho (v) que é pai de outro nodo vermelho (z), enquanto que o correto é ter dois nodos vermelhos como irmãos. Para consertar este problema, executamos uma ***reestruturação trinodo*** de T. A reestruturação trinodo é feita pela operação restructure(z), que consiste nos passos a seguir (veja novamente a Figura 3.27; esta operação também foi discutida na Seção 3.2):

- Pegue o nodo z, seu pai v e seu avô u e temporariamente troque seus rótulos para a, b e c, da esquerda para a direita, de maneira que a, b e c sejam visitados nesta ordem em um caminhamento interfixado.

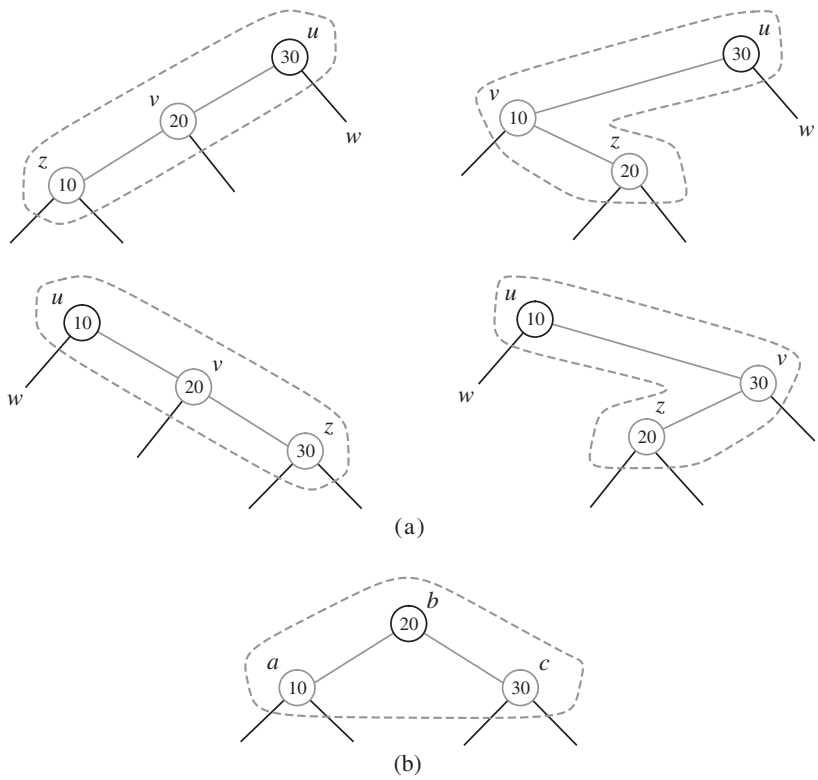


Figura 3.27 Restruturando uma árvore vermelho-preta para consertar um vermelho duplo: (a) as quatro configurações para u, v e z antes da reestruturação; (b) após a mesma.

- Substitua o avô u pelo nodo rotulado b e faça os nodos a e c serem os filhos de b, mantendo os relacionamentos interfixados inalterados.

Após executar a operação restructure(z), colorimos b de preto e a e c de vermelho. Desta forma a reestruturação elimina o problema do vermelho duplo.

Caso 2: o irmão *w* de *v* é vermelho (veja a Figura 3.28). Neste caso, o vermelho duplo indica um *overflow* na árvore (2,4) T' correspondente. Para corrigir este problema, executamos o equivalente a uma operação de divisão. Isto é, ***trocamos as cores***: colorimos v e w de preto e seu pai u de vermelho (a menos que u seja a raiz, caso em que será colorida de preto). É possível que, após a troca de cores, o problema do vermelho duplo reapareça, embora mais acima na árvore T, uma vez que u pode ter um pai vermelho. Se o problema do vermelho duplo reaparecer em u, então repetimos as considerações para os dois casos sobre u. Conseqüentemente, a troca de cores pode tanto eliminar o problema do vermelho duplo no nodo z como propagá-lo para o avô u de z. Continuamos subindo por T, executando a troca de cores até finalmente resolver o problema do vermelho duplo (usando troca de cores ou uma reestruturação trinodo). Conseqüentemente, o número de trocas de cores causadas por uma inserção não é maior que a metade da altura da árvore T, ou seja, não mais que $\log(n + 1)$ pelo Teorema 3.5.

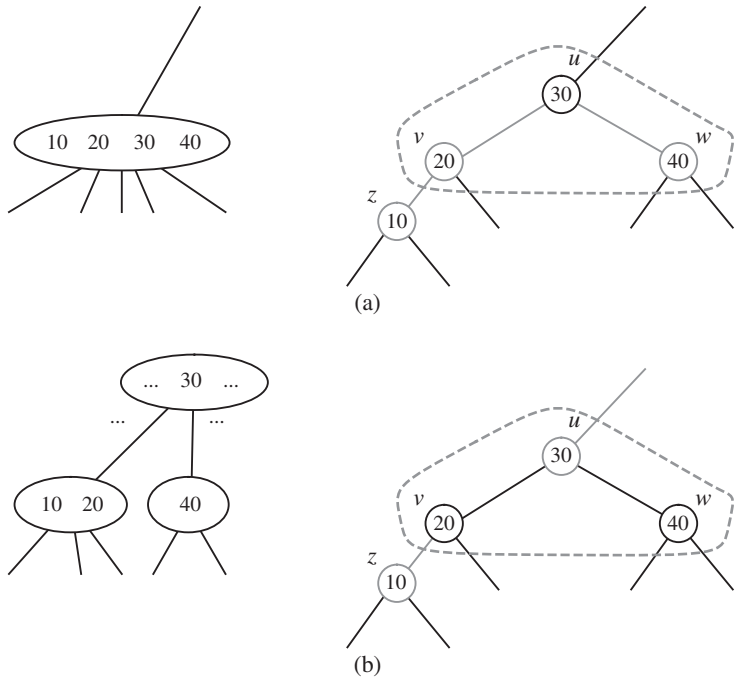


Figura 3.28 Trocando cores para remediar o problema do vermelho duplo: (a) antes da troca de cores e o 5-nodo correspondente na árvore (2,4) associada antes da divisão; (b) depois da troca de cores (e os nodos correspondentes na árvore (2,4) associada após a divisão).

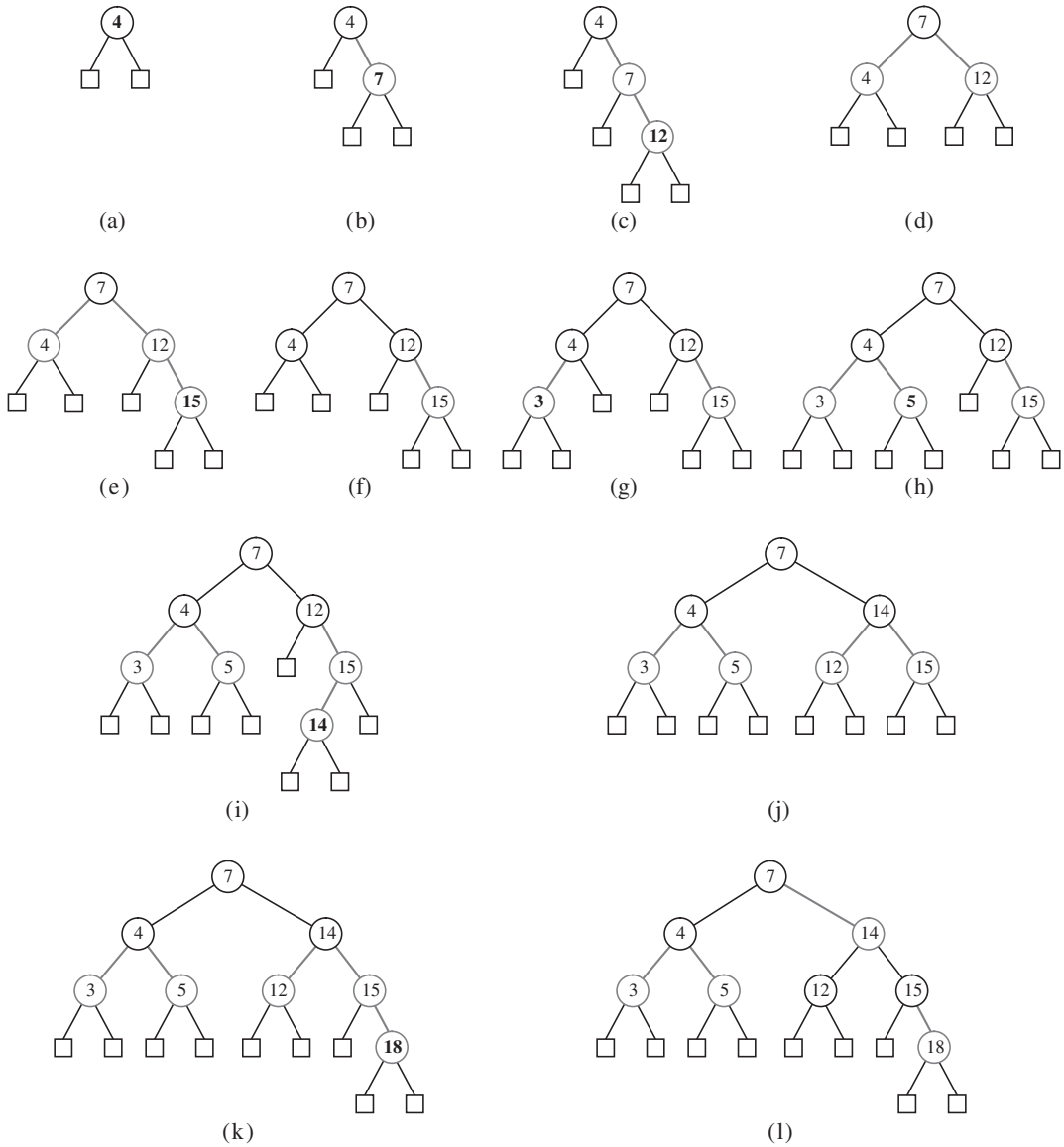


Figura 3.29 Seqüência de inserções em uma árvore vermelho-preta: (a) árvore inicial; (b) inserção de 7; (c) inserção de 12, que causa um vermelho duplo; (d) após a reestruturação; (e) inserção de 15, causando um vermelho duplo; (f) após a troca de cores (a raiz permanece preta); (g) inserção de 3; (h) inserção de 5; (i) inserção de 14, causando um vermelho duplo; (j) após a reestruturação; (k) inserção de 18, causando um vermelho duplo; (i) após a troca de cores. (continua na Figura 3.30.)

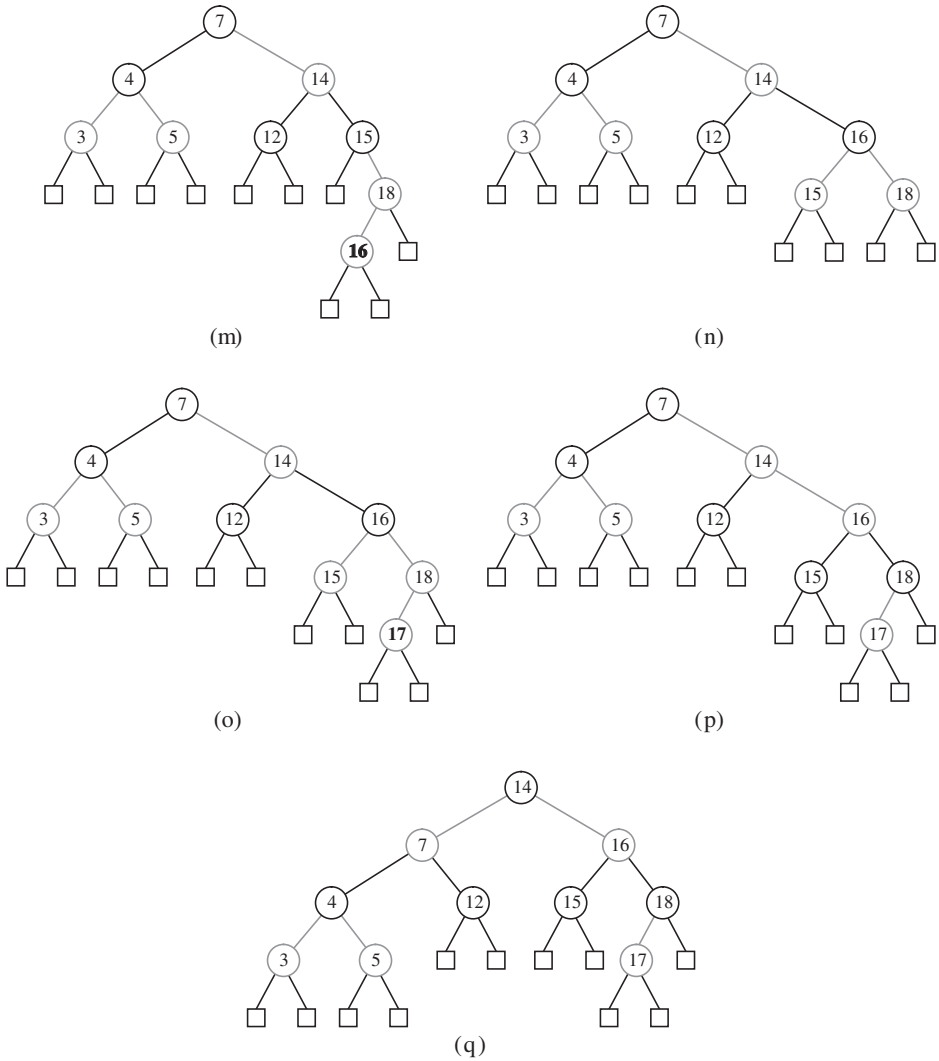


Figura 3.30 Seqüência de inserções em uma árvore vermelho-preta (continuação da Figura 3.29): (m) inserção de 16, causando um vermelho duplo; (n) após a reestruturação; (o) inserção de 17, causando um vermelho duplo; (p) após a troca de cores existe novamente um vermelho duplo a ser tratado por reestruturação; (q) após a reestruturação.

As Figuras 3.29 e 3.30 apresentam uma seqüência de operações de inserção em uma árvore vermelho-preta.

Os casos de inserção implicam uma propriedade interessante das árvores vermelho-pretas. Ou seja, uma vez que o a ação do Caso 1 elimina o problema do vermelho duplo com uma reestruturação trinodo e a ação do Caso 2 não executa operações de reestruturação, no máximo uma reestruturação será necessária por operação de inserção em árvore vermelho-preta. Por esta análise, e pelo fato de que uma reestruturação ou troca de cores levar tempo $O(1)$, temos o seguinte:

Teorema 3.6: *a inserção de um elemento chave em uma árvore vermelho-preta que armazena n itens pode ser feita em tempo $O(\log n)$ e requer no máximo $O(\log n)$ trocas de cores e uma reestruturação trinodo (uma operação* restructure*).*

Remoção em uma árvore vermelho-preta

Suponhamos agora que somos chamados a remover um item com chave k de uma árvore vermelho-preta T. A remoção de tal item se inicia como no caso de uma árvore de pesquisa binária (Seção 3.1.5). Inicialmente pesquisamos o nodo u que armazena tal item. Se o nodo u não tiver filhos externos, encontramos o nodo interno v que segue u em um caminhamento interfixado sobre T, movemos o item em v para u e efetivamos a remoção em v. Conseqüentemente, podemos considerar apenas a remoção de um item com chave k armazenado em um nodo v com filho externo w. Da mesma forma que para as inserções, devemos ter em mente a correspondência entre a árvore vermelho-preta T e a árvore (2,4) T' associada (e o algoritmo de remoção para T').

Para remover o item com chave k de um nodo v de T com um filho externo w fazemos como segue. Seja r o irmão de w e x o pai de v. Removemos os nodos v e w e fazemos r filho de x. Se v era vermelho (logo r é preto) ou r é vermelho (então v era preto), colorimos r de preto e é o suficiente. Se, por outro lado, r é preto e v era preto, então, para preservar a propriedade da profundidade, atribuímos a r um colorido ***duplo preto*** fictício. Agora temos uma violação de cor chamada de problema do duplo preto. Um duplo preto em T indica um *underflow* na árvore (2,4) T' correspondente. Lembramos que x é o pai do nodo com duplo preto r. Para contornar o problema do duplo preto em r, consideramos três casos.

Caso 1: o irmão *y* de *r* é preto e tem um filho vermelho *z* (veja a Figura 3.31). Resolver este caso corresponde a uma operação de transferência na árvore (2,4) T'. Executamos uma ***reestruturação trinodo*** usando a operação restructure(z). Lembramos que a operação restructure(z) considera o nodo z, seu pai y e o avô x, rotula os mesmos da esquerda para a direita como a, b e c e substitui x pelo nodo rotulado b, tornando-o pai dos outros dois (veja também a descrição de restructure na Seção 3.2). Colorimos a e c de preto, atribuímos a b a cor anterior de x e colorimos r de preto. Esta reestruturação trinodo elimina o problema do preto duplo. Conseqüentemente, neste caso, no máximo uma reestruturação é executada em uma operação de remoção.

Caso 2: o irmão *y* de *r* é preto e os dois filhos de *y* são pretos (veja as Figuras 3.32 e 3.33). A resolução deste caso corresponde a uma operação de fusão na árvore

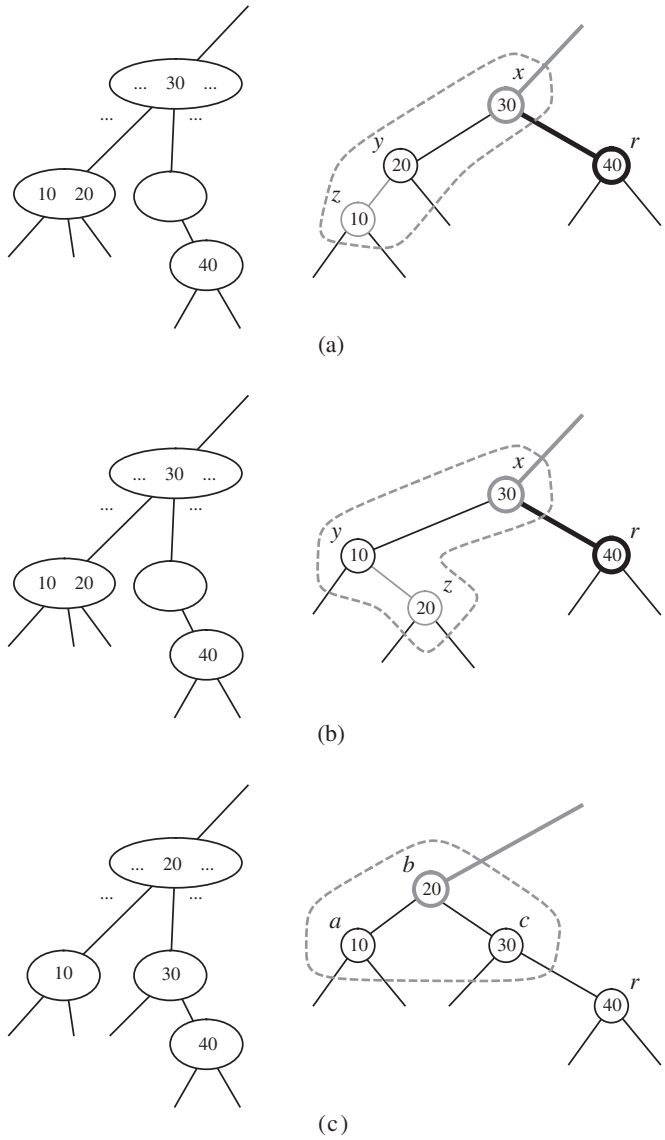


Figura 3.31 Reestruturação de uma árvore vermelho-preta para remediar o problema do duplo preto: configurações (a) e (b) antes da reestruturação, com *r* sendo um filho da direita juntamente com os nodos associados na árvore (2,4) correspondentes antes da transferência (duas outras configurações simétricas são possíveis com *r* sendo o filho da esquerda; configuração (c) após a reestruturação e os nodos associados na árvore (2,4) correspondentes após a transferência. A cor cinza do nodo *x* nas partes (a) e (b) e para o nodo *b* na parte (c) denotam o fato de que este nodo pode ser colorido tanto de vermelho como de preto.

(2,4) correspondente a T'. ***Recolorimos***; colorimos r de preto, y de vermelho e, se x é vermelho, colorimos o mesmo de preto (Figura 3.32); caso contrário, colorimos x com um preto duplo (Figura 3.33). Em conseqüência, após alterarmos as cores, o problema do preto duplo pode reaparecer no pai x de r (veja a Figura 3.33). Ou seja, a troca de cores elimina o problema do preto duplo ou propaga o mesmo para os pais do nodo corrente. Desta forma, uma vez que o Caso 1 executa uma reestruturação trinodo e pára (e, como veremos logo, o Caso 3 é similar), o número de alterações de cor causadas por um remoção não é maior do que $\log(n + 1)$.

Caso 3: o irmão *y* de *r* é vermelho (veja a Figura 3.34). Neste caso, executamos uma operação de ***ajuste***, como segue. Se y é o filho da direita de x, faça z ser o filho da direita de y; caso contrário, faça z ser o filho da esquerda de y. Execute a operação de reestruturação trinodo, restructure(z), que torna y o pai de x. Pinte y de preto e x de vermelho. Um ajuste corresponde à escolha de uma representação diferente de um 3-nodo em uma árvore (2,4) T'. Após a operação de ajuste, o irmão

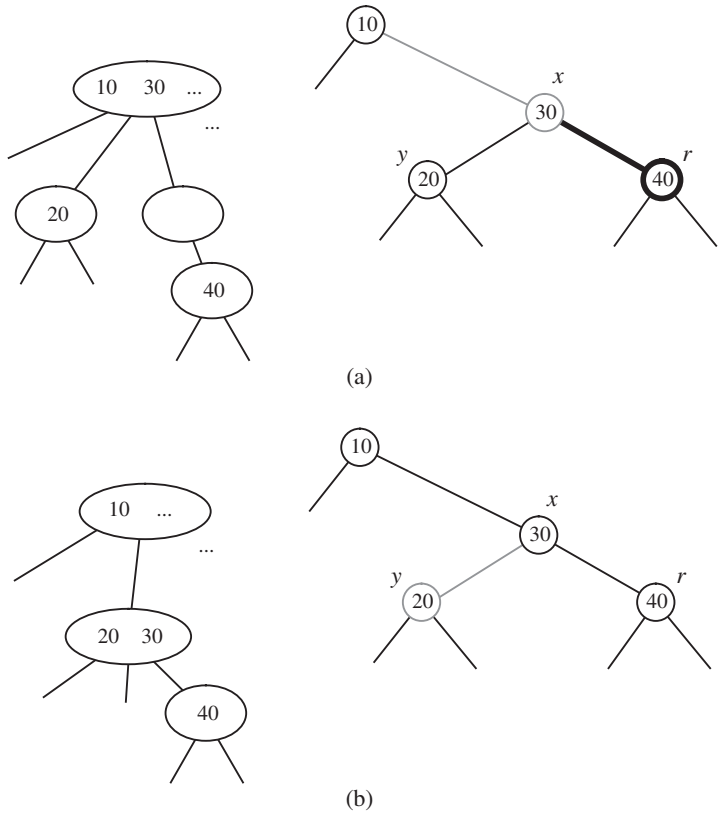


Figura 3.32 Alterando as cores de uma árvore vermelho-preta para consertar o problema do preto duplo: (a) antes da alteração de cores e os nodos correspondentes na árvore (2,4) associada antes da fusão (outras configurações semelhantes são possíveis); (b) após a troca de cores e nodos correspondentes na árvore (2,4) associada após a fusão.

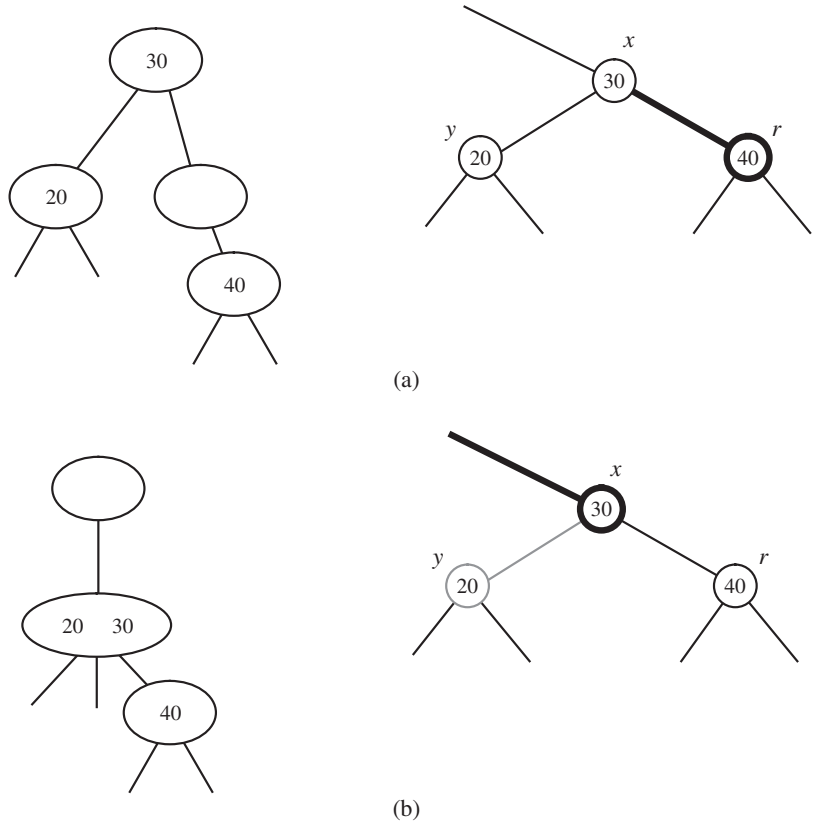


Figura 3.33 Troca de cores de uma árvore vermelho-preta que propaga o problema do preto duplo: (a) configuração antes da troca de cores e os nodos correspondentes na árvore (2,4) associada antes da fusão (outras configurações semelhantes são possíveis); (b) configuração após a troca de cores e nodos correspondentes na árvore (2,4) associada após a fusão.

de r é preto, e tanto o Caso 1 como o Caso 2 se aplicam com um significado diferente para x e y. Observemos que, se o Caso 2 se aplica, o problema do preto duplo não pode aparecer novamente. Desta forma, para completar o Caso 3, efetuamos uma ou mais aplicações tanto do Caso 1 como do Caso 2, e pronto. Por esta razão, pelo menos um ajuste é feito em uma operação de remoção.

A partir da descrição do algoritmo, vemos que a atualização de uma árvore, necessária após uma remoção, envolve um avanço para cima na árvore T, enquanto executamos uma quantidade constante de trabalho (em uma reestruturação, troca de cores ou ajuste) por nodo. Assim, uma vez que qualquer alteração feita em um nodo de T durante este avanço árvore acima leva tempo $O(1)$ (porque afeta um número constante de nodos), temos o seguinte:

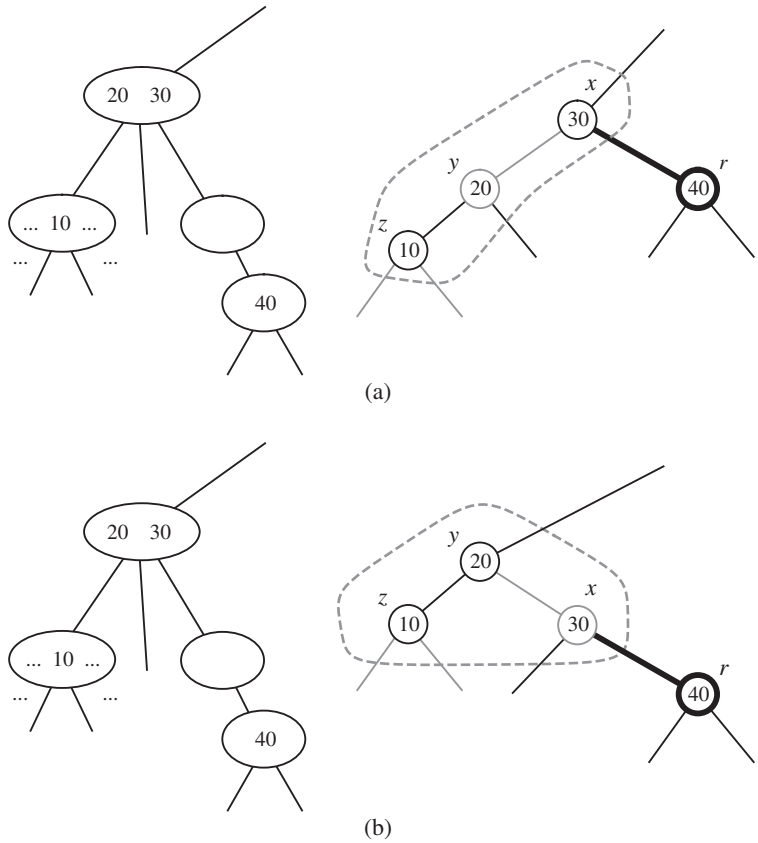


Figura 3.34 Ajuste de uma árvore vermelho-preta na ocorrência do problema de um preto duplo: (a) configuração antes do ajuste e nodos correspondentes na árvore (2,4) associada (uma configuração simétrica é possível); (b)configuração após o ajuste com os mesmos nodos correspondentes na árvore (2,4) associada.

Teorema 3.7: *o algoritmo para remoção de um item de uma árvore vermelho-preta com n itens leva tempo O*(log *n*) *e executa O*(log *n*) *mudanças de cor e, no mínimo, um ajuste e mais uma reestruturação trinodo adicional. Desta forma, executa pelo menos duas operações* restructure.

Nas Figuras 3.35 e 3.36, indicamos uma seqüência de operações de remoção em uma árvore vermelho-preta. Demonstramos restruturações do tipo do Caso 1 na Figura 3.35c e d. Apresentamos troca de cores do tipo do Caso 2 em vários locais das Figuras 3.35 e 3.36. Finalmente, na Figura 3.36i e j, mostramos um exemplo de ajuste do tipo previsto no Caso 3.

Desempenho de uma árvore vermelho-preta

A Tabela 3.37 resume os tempos de execução das principais operações do dicionário implementado usando uma árvore vermelho-preta. As justificativas para estes limites, apresentamos na Figura 3.38.

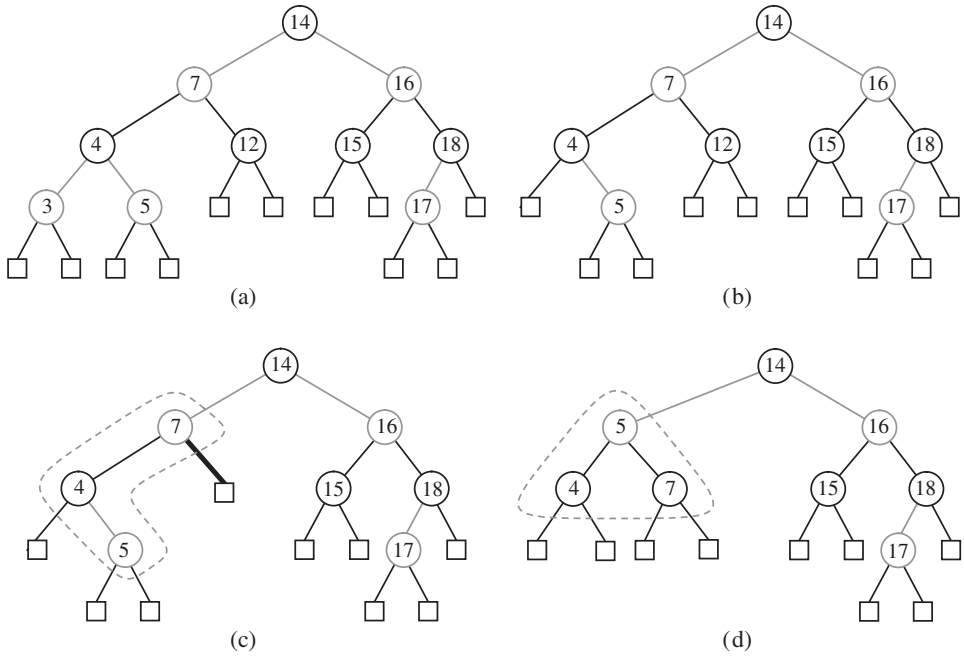


Figura 3.35 Seqüência de remoções de uma árvore vermelho-preta: (a) árvore inicial; (b) remoção de 3; (c) remoção de 12, causando um duplo preto (tratado por reestruturação); (d) após a restruturação.

Desta forma, uma árvore vermelho-preta obtém tempo de execução logarítmico para o pior caso tanto para pesquisa como para atualização em um dicionário. A estrutura de árvore vermelho-preta é ligeiramente mais complicada do que a da árvore (2,4) correspondente. Apesar disso, uma árvore vermelho-preta tem a vantagem conceitual de requerer apenas um número constante de reestruturações trinodo para restaurar o balanceamento após uma atualização.

3.4 *Splay trees*

A última estrutura de pesquisa balanceada que será discutida neste capítulo é a ***splay tree***. Esta estrutura é conceitualmente diferente das árvores balanceadas que mostramos anteriormente (AVL, vermelho-preta e árvore (2,4)), pois uma *splay tree* não usa regras explícitas para forçar seu balanceamento. Em vez disso, aplica-se uma certa operação de movimentação em direção à raiz chamada de **espalhamento** (ou *splaying*) após cada acesso, mantendo a árvore de pesquisa balanceada sob uma perspectiva amortizada. A operação de espalhamento é executada no nodo mais baixo, x, encontrado durante uma inserção, deleção ou mesmo pesquisa. O surpreendente a respeito do espalhamento é que ele nos permite garantir tempos de execução amortizados para inserções, deleções e pesquisa que são logarítmicos. A estrutura de uma ***splay tree*** é a de uma simples árvore de pesquisa binária T. De fato, não existe altura adicional, balanceamento ou rótulos coloridos associados com os nodos desta árvore.

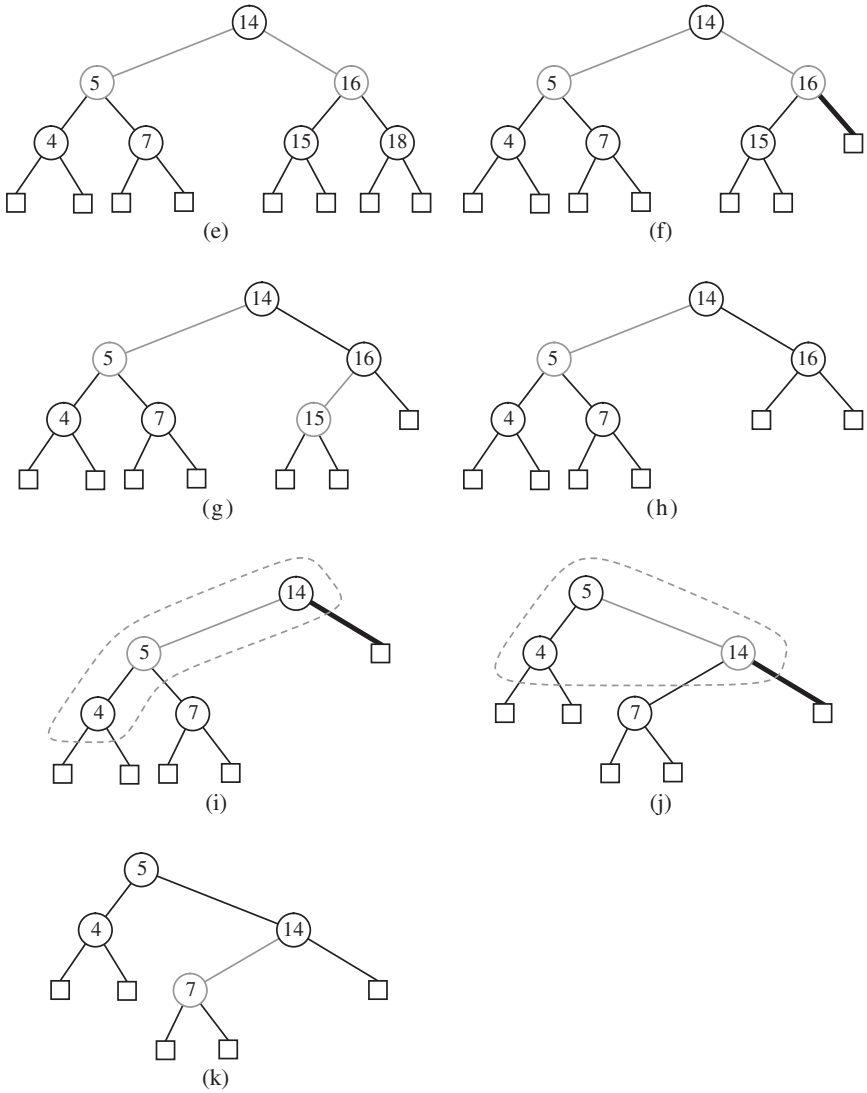


Figura 3.36 Seqüência de remoções em uma árvore vermelho-preta (continuação): (e) remoção de 17; (f) remoção de 18, causando um duplo preto (tratado por troca de cores); (g) após a troca de cores; (h) remoção de 15; (i) remoção de 16, causando um duplo preto (tratado por ajuste); (j) após o ajuste, o duplo preto necessita ser tratado por troca de cores; (k) após a troca de cores.

Operação	Tempo
size, isEmpty	$O(1)$
findElement, insertItem, removeElement	$O(\log n)$
findAllElements, removeAllElements	$O(\log n + s)$

Tabela 3.37 Desempenho de um dicionário de n elementos implementado usando uma árvore vermelho-preta onde s denota o tamanho dos iteradores retornados por findAllElements e removeAllElements. O espaço utilizado é $O(n)$.

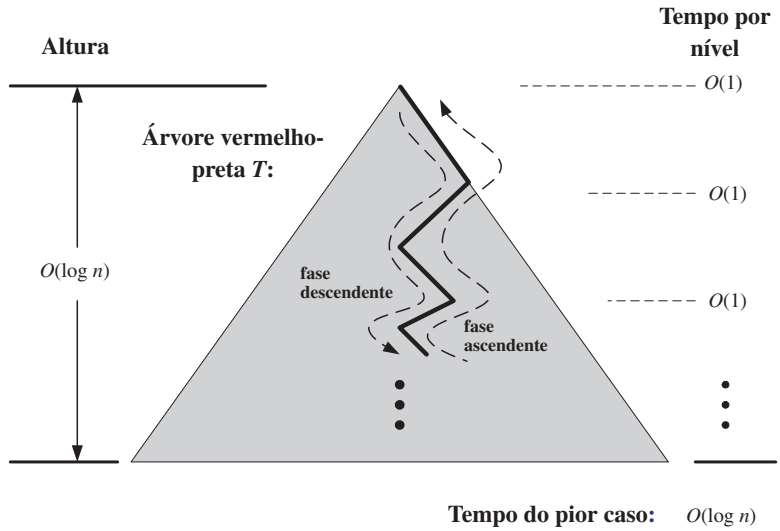


Figura 3.38 Demonstração do tempo de execução de pesquisa e atualização em uma árvore vermelho-preta. O desempenho de tempo é $O(1)$ por nível, quebrado em uma fase descendente, que tipicamente envolve pesquisa, e uma fase ascendente, que tipicamente envolve troca de cores e execução de reestruturações trinodo locais (rotações).

3.4.1 Espalhamento

Dado um nodo interno x de uma árvore de pesquisa binária T, aplicamos o **espalhamento** (***splaying***) sobre x movendo x para a raiz de T através de uma seqüência de reestruturações. As reestruturações específicas que executamos são importantes; porém, uma seqüência de reestruturações não é suficiente para mover x para a raiz de T. A operação específica que executamos para mover x para cima depende da posição relativa de x, de seu pai y e (se existir) do avô z de x. Existem três casos para considerar.

Zigue-zigue: o nodo x e seu pai y são filhos da direita ou esquerda (veja a Figura 3.39). Substituímos z por x transformando y em filho de x e, z em filho de y, enquanto mantemos os relacionamentos interfixados dos nodos de T.

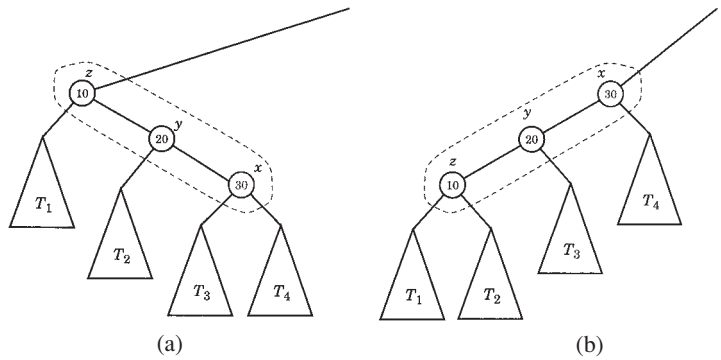


Figura 3.39 Zigue-zigue: (a) antes; (b) depois. Existe outra configuração simétrica onde x e y são filhos da esquerda.

Zigue-zague: ou x ou y é um filho da esquerda, e o outro é um filho da direita (veja a Figura 3.40). Neste caso, substituímos z por x e fazemos x ter como filhos os nodos y e z, enquanto mantemos os relacionamentos interfixados dos nodos de T.

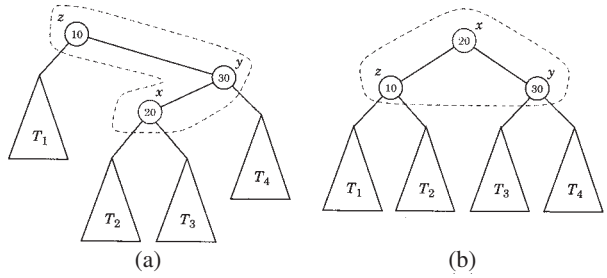


Figura 3.40 Zigue-zague: (a) antes; (b) depois. Existe outra configuração simétrica onde x é o filho da direita e y é o filho da esquerda.

Zigue: x não tem avô (ou não estamos considerando o avô de x por alguma razão) (veja a Figura 3.41). Neste caso, rotamos x ao redor de y, fazendo o filho de x ser o nodo y, e um dos primeiros filhos de x ser w, de maneira a manter os relacionamentos relativos interfixados entre os nodos de T.

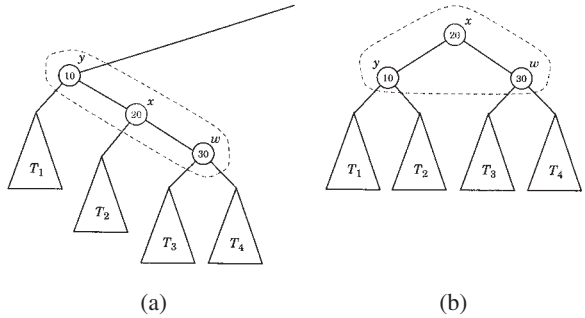
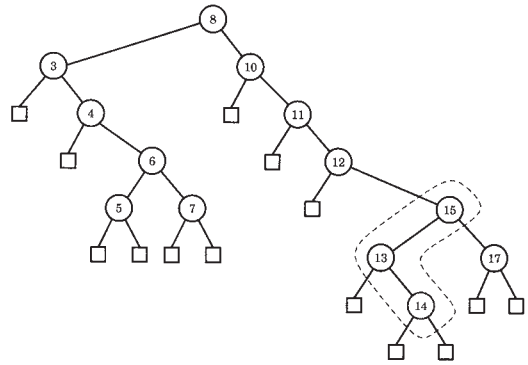


Figura 3.41 Zigue: (a) antes; (b) depois. Existe outra configuração simétrica onde x e w são filhos da esquerda.

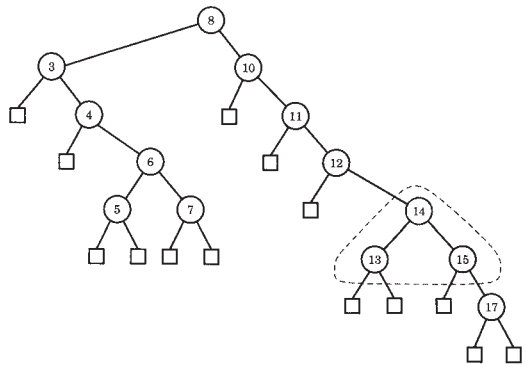
Executamos um zigue-zigue ou zigue-zague quando x tem um avô, e executamos um zigue quando x tem pai, mas não têm avô. A etapa de espalhamento consiste na repetição destas reestruturações sobre x até que x se torne a raiz de T. Observamos que esta não é a mesma seqüência simples de rotações que leva x à raiz. Um exemplo do espalhamento de um nodo é apresentado nas Figuras 3.42 e 3.43.

Depois de um zigue-zigue ou de um zigue-zague, a profundidade de x diminui em dois, e, depois de um zigue, diminui em um. Assim, se x tem profundidade d, a operação de espalhamento sobre x consiste em uma seqüência de $\lfloor d/2 \rfloor$ zigue-zigues e/ou zigue-zagues, mais um zigue final se d for ímpar. Dado que um simples zigue-zigue ou zigue-zague afeta um número constante de nodos, ele pode ser executado em tempo $O(1)$. Assim, a operação de espalhamento sobre um nodo x em uma árvore binária T consome tempo $O(d)$, onde d corresponde à profundidade de x em T. Em outras palavras, o tempo para executar uma etapa de espalhamento para um nodo x é assintótica-

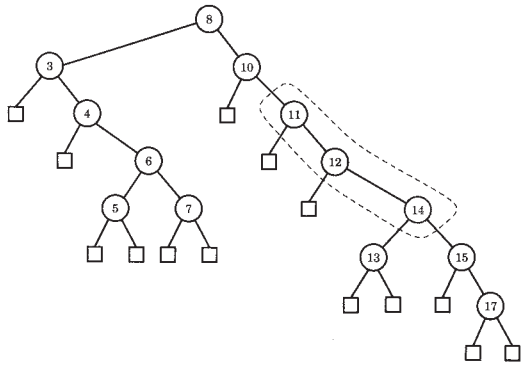
mente o mesmo que o tempo necessário para encontrar um nodo em uma pesquisa *top-down* partindo da raiz de T.



(a)

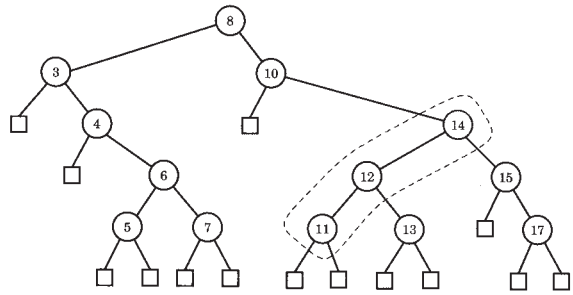


(b)

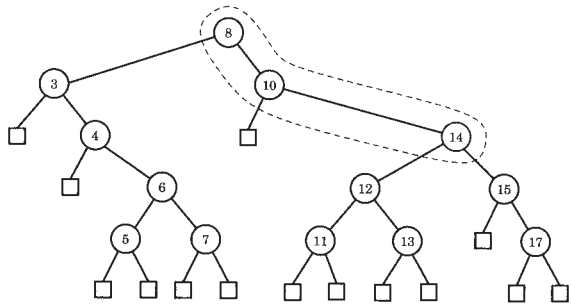


(c)

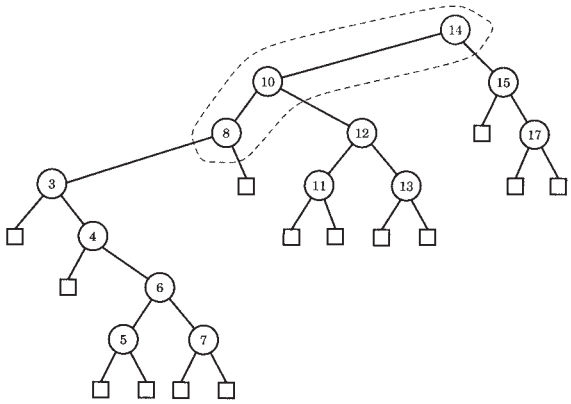
Figura 3.42 Exemplo de operação de espalhamento sobre um nodo: (a) aplicando o espalhamento sobre o nodo que armazena 14 inicia por um zigue-zague; (b) depois do zigue-zague; (c) o próximo passo é um zigue-zigue.



(d)



(e)



(f)

Figura 3.43 Exemplo da aplicação de um espalhamento sobre um nodo (continuação da Figura 3.42): (d) após o zigue-zigue; (e) o próximo passo é outro zigue-zigue; (f) após o zigue-zigue.

Quando aplicar o espalhamento

As regras que ditam quando um espalhamento é executado são as seguintes:

- Quando se procura por uma chave k, se k é encontrado em um nodo x, aplicamos espalhamento sobre x; caso contrário, aplicamos o espalhamento sobre o pai do nodo externo no qual a pesquisa terminou sem sucesso. Por exemplo, o espalhamento nas Figuras 3.42 e 3.43 pode ser aplicado após a pesquisa bem-sucedida pelo número 14 ou após a pesquisa malsucedida pelo número 14,5.

- Quando se insere uma chave k, aplicamos o espalhamento sobre o nodo interno recém-criado onde k é inserido. Por exemplo, o espalhamento das Figuras 3.42 e 3.43 pode ser executado se 14 for a chave recém-inserida. Apresentamos uma seqüência de inserções em uma *splay tree* na Figura 3.44.
- Quando se elimina uma chave k, aplicamos o espalhamento sobre o pai do nodo w que é removido, isto é, w pode ser tanto o nodo que armazena k como um de seus descendentes. (Lembre-se do algoritmo de deleção para árvores binárias de pesquisa fornecido na Seção 3.1.2). Um exemplo de aplicação de espalhamento após uma deleção é apresentado na Figura 3.45.

No pior caso, o tempo total de pesquisa, inserção ou deleção em uma *splay tree* de altura h é $O(h)$, uma vez que o nodo sobre o qual se aplica o espalhamento pode ser o mais ao fundo da árvore. Além disso, é possível para h ser $\Omega(n)$, como mostrado na Figura 3.44. Assim, considerando o pior caso, uma *splay tree* não é uma estrutura de dados muito atraente.

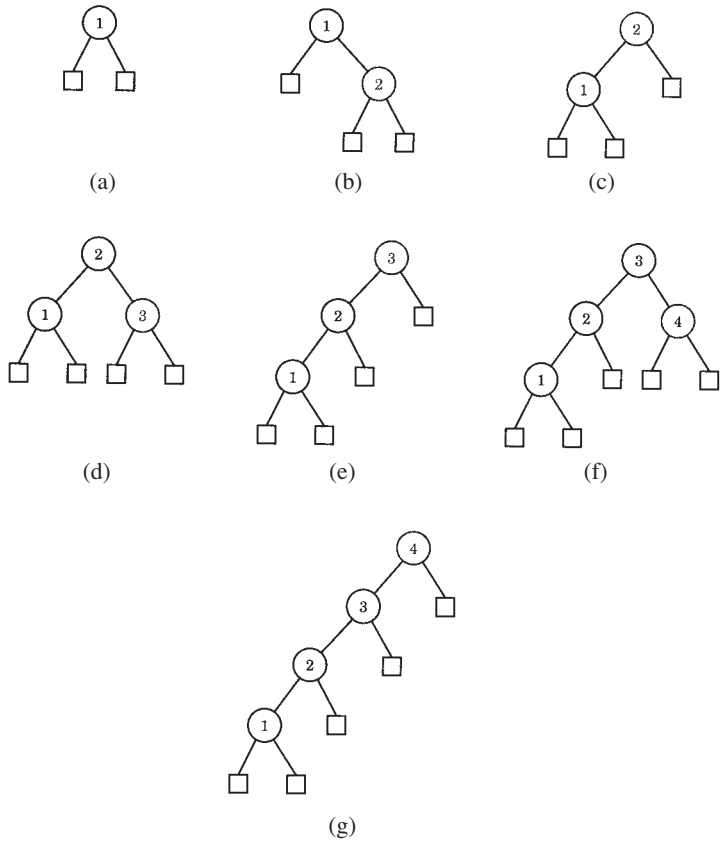


Figura 3.44 Seqüência de inserções em uma *splay tree*: (a) árvore inicial; (b) após inserir 2; (c) após aplicar um espalhamento; (d) após inserir 3; (e) após aplicar um espalhamento; (f) após inserir 4; (g) após aplicar um espalhamento.

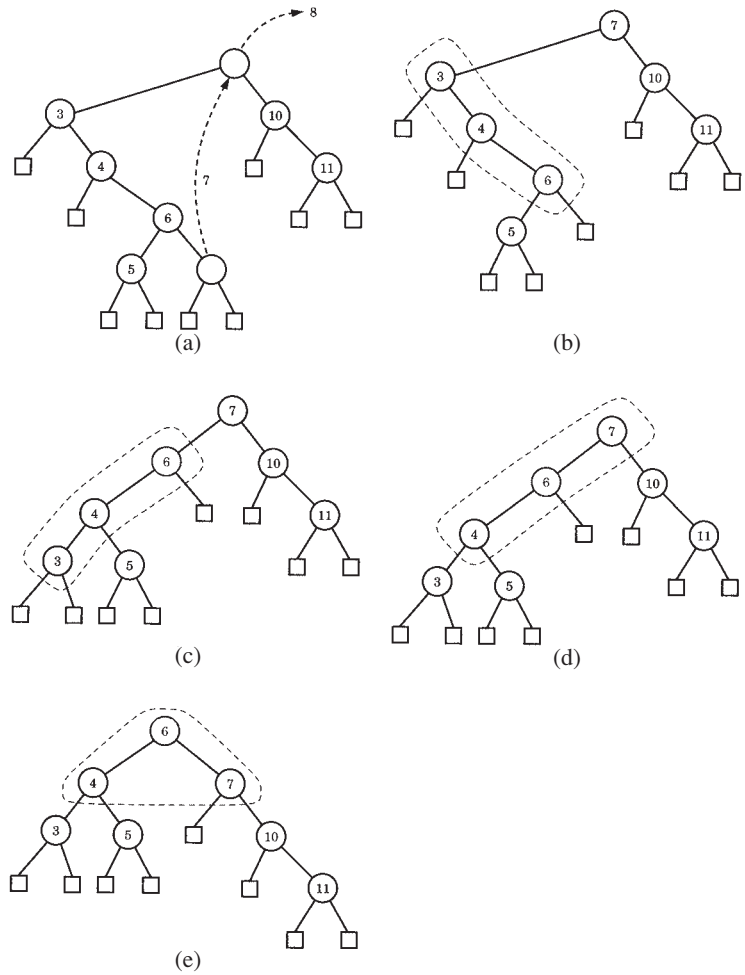


Figura 3.45 Eliminação de um nodo de uma *splay tree*: (a) a deleção do valor 8 do nodo *r* é executada movendo-se para *r* a chave do nodo interno mais à direita *v* da subárvore esquerda de *r,* eliminando *v* e aplicando o espalhamento sobre o pai *u* de *v*; (b) para aplicar o espalhamento sobre *u*, inicia-se com um zigue-zigue; (c) após o zigue-zigue; (d) o próximo passo é um zigue; (e) após o zigue.

3.4.2 Análise amortizada da operação de espalhamento

Apesar de seu desempenho pobre no pior caso, uma *splay tree* se comporta bem em uma perspectiva amortizada. Isto é, em uma seqüência variada de pesquisas, inserções e remoções, cada operação tira vantagem do tempo logarítmico. Observamos que o tempo para executar uma pesquisa, inserção ou remoção é proporcional ao tempo do espalhamento correspondente; assim, em nossa análise, consideraremos apenas o tempo de espalhamento.

Seja T uma *splay tree* com n chaves, e seja v um nodo de T. Definimos o ***tamanho*** $n(v)$ como o número de nodos na subárvore enraizada em v. Note que o tamanho de um nodo interno é um a mais que a soma dos tamanhos de seus dois filhos. Definimos a

colocação $r(v)$ de um nodo v como sendo o logaritmo na base 2 do tamanho de v, isto é, $r(v) = \log n(v)$. Claramente, a raiz de T tem o tamanho máximo igual a $2n + 1$ e a colocação máxima, $\log(2n + 1)$, enquanto que cada nodo externo tem tamanho 1 e colocação 0.

Usaremos ciberdólares para pagar pelo trabalho que executamos em uma operação de espalhamento sobre um nodo x de T e assumimos que um zigue custa um ciberdólar, enquanto que um zigue-zigue ou zigue-zague custam dois ciberdólares. Manteremos uma conta virtual, poupando ciberdólares, em cada nodo interno de T. Observe que esta conta existe apenas com o propósito de fazermos nossa análise amortizada e não necessita ser incluída na implementação de uma *splay tree T*. Quando executamos uma operação de espalhamento, pagamos uma certa quantidade de ciberdólares (o valor exato será determinado mais tarde). Identificamos três casos:

- Se o pagamento for igual ao esforço para aplicar o espalhamento, então usamos todos os ciberdólares para pagar a operação.
- Se o pagamento for maior que o esforço para aplicar o espalhamento, depositamos o excesso nas contas de vários nodos.
- Se o pagamento for menor que o esforço para aplicar o espalhamento, fazemos retiradas das contas de vários nodos para cobrir o déficit.

Mostraremos que o pagamento de $O(\log n)$ ciberdólares por operação é suficiente para manter o sistema operando, ou seja, para garantir que cada nodo mantenha uma conta não negativa. Usamos um esquema no qual transferências são feitas entre as contas dos nodos para garantir que sempre haverá ciberdólares suficientes para pagar o esforço de espalhamento quando necessário. Também mantemos a seguinte invariante:

Antes e depois de um espalhamento, cada nodo v ***de T tem*** $r(v)$ ***ciberdólares***

Observamos que esta invariante não requer que doemos uma árvore vazia com ciberdólares. Seja $r(T)$ a soma das colocações de todos os nodos de T. Para preservar o invariante após uma operação de espalhamento, devemos fazer um pagamento igual ao esforço de espalhamento mais o total alterado em $r(T)$. Referimo-nos a um simples zigue, zigue-zigue ou zigue-zague em uma operação de espalhamento como sendo um ***passo intermediário*** da operação de espalhamento. Da mesma forma, denotamos a colocação de um nodo v de T antes e depois de um passo intermediário com $r(v)$ e $r'(v)$, respectivamente. O lema a seguir nos fornece um limite superior para as alterações sobre $r(T)$ em conseqüência de um único passo intermediário.

Lema 3.8: *seja* δ *a variação de* $r(T)$ *causada por um único passo intermediário de splay (um zigue, zigue-zigue ou zigue-zague) sobre um nodo x de uma splay tree T. Teremos o que segue:*

- $\delta \leq 3(r'(x) - r(x))$ *se o passo intermediário for um zigue-zigue ou zigue-zague.*
- $\delta \leq 3(r'(x) - r(x))$ *se o passo intermediário for um zigue.*

Prova: devemos fazer uso de um fato matemático (veja Apêndice A):

Se $a > 0$, $b > 0$ e $c > a + b$, então

$$\log a + \log b \leq 2 \log c - 2. \tag{3.6}$$

Consideremos a alteração em $r(T)$ causada por cada passo intermediário.

Zigue-zigue (lembre-se da Figura 3.39): uma vez que o tamanho de cada nodo é maior em uma unidade que o tamanho de seus dois filhos, observe que apenas as colocações de x,y e z se alteram em uma operação zigue-zigue, onde y é o pai de x e z é pai de y. Além disso, $r'(x) = r(z)$, $r'(y) \leq r'(x)$, e $r(y) \geq r(x)$. Logo,

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(x) + r'(z) - 2r(x).\end{aligned} \tag{3.7}$$

Observe que $n(x) + n'(z) \leq n'(x)$. Assim, por 3.6, $r(x) + r'(z) \leq 2r'(x) - 2$. Isto é,

$$r'(z) \leq 2r'(x) - r(x) - 2$$

Essa desigualdade e 3.7 implicam

$$\begin{aligned}\delta &\leq r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2.\end{aligned}$$

Zigue-zague (lembre-se da Figura 3.40): novamente, pela definição de tamanho e colocação, apenas as colocações de x, y e z se alteram, onde y indica o pai de x e z indica o pai de y. Também, $r'(x) = r(z)$ e $r(x) \leq r(y)$. Assim,

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(y) + r'(z) - 2r(x).\end{aligned} \tag{3.8}$$

Observe que $n'(y) + n'(z) \leq n'(x)$. Assim, por 3.6, $r'(y) + r'(z) \leq 2r'(x) - 2$. Esta desigualdade e 3.8 implicam

$$\begin{aligned}\delta &\leq 2r'(x) - 2 - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2.\end{aligned}$$

Zigue (lembre-se da Figura 3.41): neste caso, apenas as colocações de x e y se alteram, onde y denota o pai de x. Também, $r'(y) \leq r(y)$ e $r'(x) \geq r'(x)$. Assim,

$$\begin{aligned}\delta &= r'(y) + r'(x) - r(y) - r(x) \\ &\leq r'(x) - r(x) \\ &\leq 3(r'(x) - r(x)).\end{aligned}$$

■

Teorema 3.9: *seja T uma splay tree com raiz t, e seja Δ a variação total de $r(T)$ causada pelo espalhamento de um nodo x na profundidade d. Teremos*

$$\Delta \leq 3(r(t) - r(x)) - d + 2.$$

Prova: aplicar um espalhamento sobre o nodo x consiste em $p = \lceil d/2 \rceil$ passos intermediários, cada um dos quais sendo um zigue-zigue ou zigue-zague, exceto no caso

do último que é um zigue se d for ímpar. Seja $r_0(x) = r(x)$ a colocação inicial de x, e para $i = 1, ..., p$, seja $r_i(x)$ a colocação de x após o i-ésimo passo intermediário e δ_i ser a variação de $r(T)$ causada pelo i-ésimo passo intermediário. Pelo Lema 3.8, a variação total Δ de $r(T)$ causada pelo espalhamento do nodo x é dada por

$$\begin{aligned}\Delta &= \sum_{i=1}^{p} \delta_i \\ &\leq \sum_{i=1}^{p} (3(r_i(x) - r_{i-1}(x)) - 2) + 2 \\ &= 3(r_p(x) - r_0(x)) - 2p + 2 \\ &\leq 3(r(t) - r(x)) - d + 2.\end{aligned}$$

■

Pelo Teorema 3.9, se fizermos o pagamento de $3(r(t) - r(x)) + 2$ ciberdólares pelo espalhamento do nodo x, teremos ciberdólares suficientes para manter a invariante, conservando $r(v)$ ciberdólares em cada nodo v de T, e pagando por todo o esforço de espalhamento, que custa d dólares. Considerando que o tamanho da raiz t é $2n + 1$, sua colocação é $r(t) = \log(2n + 1)$. Além disso, temos $r(x) < r(t)$. Assim, o pagamento a ser feito pelo espalhamento é $O(\log n)$ ciberdólares. Para completar nossa análise, temos de calcular o custo de manutenção da invariante quando um nodo é inserido ou removido.

Quando inserimos um novo nodo v em uma *splay tree* com n chaves, as colocações de todos os ancestrais de v aumentam. Na verdade, sejam $v_0, v_1, ..., v_d$ os ancestrais de v, onde $v_0 = v$, v_i é o pai de v_{i-1} e v_d é a raiz. Para $i = 1, ..., d$ faça $n'(v_i)$ e $n(v_i)$ ser o tamanho de v_i antes e depois da inserção, respectivamente, e faça $r'(v_i)$ e $r(v_i)$ serem a colocação de v_i antes e depois da inserção, respectivamente. Teremos

$$n'(v_i) = n(v_i) + 1.$$

Além disso, uma vez que $n(v_i) + 1 \leq n(v_{i+1})$, para $i = 0,1, ..., d - 1$, temos o seguinte para cada i neste intervalo:

$$r'(v_i) = \log(n'(v_i)) = \log(n(v_i) + 1) \leq \log(n'(v_{i+1}) = r(v_{i+1}).$$

Assim, a variação total de $r(T)$ causada pela inserção é

$$\begin{aligned}\sum_{i=1}^{d} (r'(v_i) - r(v_i)) &\leq r'(v_d) + \sum_{i=1}^{d-1} (r(v_{i+1})) - r(v_i) \\ &= r'(v_d) - r(v_0) \\ &\leq \log(2n+1).\end{aligned}$$

Logo, o pagamento de $O(\log n)$ ciberdólares é suficiente para manter a invariante quando um novo nodo é inserido.

Quando deletamos um nodo v de uma *splay tree* de n chaves, as colocações de todos os ancestrais de v são diminuídas. Assim, a variação total de $r(T)$ causada pela deleção é negativa, e não necessitamos fazer pagamentos para manter a invariante. Sendo assim, podemos resumir nossa análise amortizada no teorema a seguir.

Teorema 3.10: *considere uma seqüência de m operações em uma splay tree, sejam pesquisa, inserção ou remoção, iniciando com uma árvore vazia sem chaves. Além disso, seja n_i o número de chaves na árvore após a operação i, e n o número total de inserções. O tempo total de execução para executar uma seqüência de operações é*

$$O\left(m + \sum_{i=1}^{m} \log n_i\right),$$

que corresponde a $O(m \log n)$.

Em outras palavras, o tempo de execução amortizado para executar uma pesquisa, inserção ou remoção em uma *splay tree* é $O(\log n)$, onde n é o tamanho da *splay tree* no tempo. Assim, uma *splay tree* pode obter desempenho logarítmico amortizado na implementação de um TAD dicionário ordenado. Este desempenho amortizado fecha com o desempenho de pior caso das árvores AVL, árvores (2,4) e árvores vermelho-pretas, mas faz isso usando apenas uma simples árvore binária que não requer nenhuma informação extra de balanceamento armazenada em seus nodos. Além disso, *splay trees* têm outras propriedades interessantes que não são compartilhadas por essas outras árvores de pesquisa balanceadas. Exploramos uma destas propriedades adicionais no teorema a seguir (que algumas vezes é chamado de teorema da "Otimização Estática" para *splay trees*).

Teorema 3.11: *considere uma seqüência de m operações sobre uma splay tree, sejam elas pesquisa, inserção ou remoção iniciando com uma árvore T sem chaves. Além disso, faça f(i) indicar o número de vezes que o item i é acessado, isto é, sua* ***freqüência****, e faça n indicar o número total de itens. Supondo que cada item é acessado pelo menos uma vez, então o tempo total de execução para executar a seqüência de operações é*

$$O\left(m + \sum_{i=1}^{n} f(i) \log(m/f(i))\right).$$

Deixamos a prova desse teorema como exercício. O ponto interessante desse teorema é que ele afirma que o tempo amortizado para acessar um item i é $O(\log(m/f(i)))$. Por exemplo, se uma seqüência de operações acessa um item i mais de $m/4$ vezes, então o tempo amortizado de cada um destes acesso é $O(1)$ quando o dicionário é implementado usando uma *splay tree*. Compare isso com o tempo $\Omega(\log n)$ necessário para acessar este item se o dicionário for implementado usando uma árvore AVL, árvore (2,4) ou árvore vermelho-preta. Assim, uma propriedade adicional de uma *splay tree* é que ela pode se adaptar às formas pelas quais os itens estão sendo acessados em um dicionário, de maneira a obter tempos de execução mais rápidos para os itens mais freqüentemente acessados.

3.5 *Skip lists*

Uma estrutura de dados interessante para a implementação eficiente de um TAD dicionário ordenado é a ***skip list***. Esta estrutura de dados faz escolhas aleatórias ao arranjar os itens de tal forma que os tempos de pesquisa e de atualização são $O(\log n)$ ***em média***.

Randomização de estruturas de dados e algoritmos

É interessante notar que a noção de complexidade de tempo médio não depende da distribuição probabilística das chaves usadas na entrada. Em vez disso, ela depende do uso de um gerador de números aleatórios na implementação das inserções para ajudar a decidir onde colocar o novo item. O tempo de execução é ponderado por todas os valores possíveis do gerador de números aleatórios usado na inserção de itens.

Por serem usados extensivamente em jogos, criptografia e simulação, métodos que geram números aleatórios são disponíveis na maioria dos computadores. Alguns métodos, chamados de ***geradores de números pseudo-aleatórios***, geram números aparentemente aleatórios de forma determinística, começando com um número inicial chamado de ***semente***. Outros métodos usam dispositivos de *hardware* para obter números aleatórios "verdadeiros". Em qualquer caso, suporemos que nosso computador tem acesso a números que sejam suficientemente aleatórios para nossa análise.

A maior vantagem do uso da ***randomização*** em estruturas de dados e em análise de algoritmos é que as estruturas e métodos resultantes são, em geral, simples e eficientes. Podemos imaginar uma estrutura de dados randomizada bastante simples, chamada *skip list*, com os mesmos tempos logarítmicos da pesquisa binária para procura. Mesmo assim, esses tempos são ***esperados*** para a *skip list*, enquanto forem tempos de ***pior caso*** para a procura com pesquisa binária em uma tabela de pesquisa. Por outro lado, *skip lists* são muito mais rápidas do que tabelas de pesquisa na atualização e alteração de valores.

Definição de *skip list*

Uma ***skip list*** S para o dicionário D consiste em uma série de listas $\{S_0, S_1,..., S_h\}$. Cada lista S_i armazena um subconjunto dos itens de D armazenados em ordem não-decrescente de chave mais dois itens com chaves especiais, indicados por $-\infty$ e $+\infty$, onde $-\infty$ é menor do que qualquer chave que possa ser inserida em D, e $+\infty$ é maior do que qualquer chave que possa ser inserida no dicionário. Adicionalmente, as listas em S satisfazem às seguintes condições:

- A lista S_0 contém cada item do dicionário D (mais os itens especiais com chaves $-\infty$ e $+\infty$).
- Para $i = 1,..., h - 1$, a lista S_i contém (além de $-\infty$ e $+\infty$) um subconjunto gerado aleatoriamente dos itens da lista S_{i-1}.
- A lista S_h contém somente $-\infty$ e $+\infty$.

Um exemplo de uma *skip list* é mostrado na Figura 3.46. É costumeiro representar uma *skip list* S com a lista S_0 na base e as listas $S_1,..., S_h$ acima dela. Também nos referimos a h como a ***altura*** da *skip list* S.

Intuitivamente, as listas são construídas de forma que S_{i+1} contenha aproximadamente um de cada dois itens de S_i. Como veremos nos detalhes do método de inserção, os itens em S_{i+1} são escolhidos aleatoriamente dentre os itens em S_i com probabilidade 1/2. Ou seja, em essência "jogamos uma moeda" para cada item em S_i e colocamos o item em S_{i+1} se a moeda der "coroa". Assim, esperamos que S_1 tenha aproximadamente $n/2$ itens, que S_2 tenha $n/4$ itens e, em geral, que S_i tenha $n/2^i$ itens. Em outras palavras, esperamos que a altura h de S seja em torno de $\log(n)$. A redução do número de itens pela metade de uma lista para a próxima não é uma propriedade explicitamente exigida das *skip lists*, no entanto. Em vez disso, é usada a randomização.

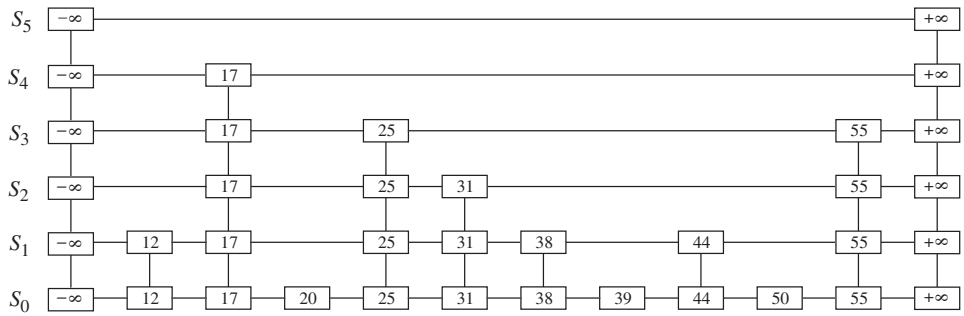


Figura 3.46 Exemplo de uma *skip list*.

Usando a abstração de posição usada para árvores e listas, vemos uma *skip list* como uma coleção bidimensional de posições arranjadas horizontalmente em ***níveis*** e verticalmente em ***torres***. Cada nível é uma *skip list* S_i, e cada torre contém posições que armazenam o mesmo item através de listas consecutivas. As posições em uma *skip list* podem ser percorridas usando-se as seguintes operações:

after(p): retorna a posição seguinte a p no mesmo nível.

before(p): retorna a posição anterior a p no mesmo nível.

below(p): retorna a posição abaixo de p na mesma torre.

above(p): retorna a posição acima de p na mesma torre.

Convencionalmente assumimos que as operações acima retornam **null** se a posição requisitada não existir. Sem entrar em detalhes, notamos que podemos implementar facilmente uma *skip list* através de uma estrutura encadeada de tal forma que os métodos descritos acima custem tempo $O(1)$, dada uma posição p na *skip list*. Uma tal estrutura encadeada é essencialmente uma coleção de h listas duplamente encadeadas alinhadas como torres, que também são listas duplamente encadeadas.

3.5.1 Pesquisa

Uma *skip list* permite algoritmos simples de pesquisa em um dicionário. De fato, todos os algoritmos de pesquisa de uma *skip list* são baseados em um elegante método SkipSearch, mostrado no Algoritmo 3.47, que recebe uma chave h e encontra o item na *skip list* S com a maior chave (que pode ser $-\infty$) que é menor ou igual a k.

Algoritmo SkipSearch(k)
  ***Entrada*:** uma chave de pesquisa k
  ***Saída*:** posição em S cujo item tem a maior chave menor ou igual a k
  Esteja p na posição mais acima e à esquerda de S (que deve ter no mínimo dois níveis).
  enquanto below(p) $\neq$ **null faça**
    $p \leftarrow$ below(p) {desce uma posição}
    enquanto key(after(p)) $\leq k$ **faça**
      Seja $p \leftarrow$ after(p) {varredura para trás}
  retorne p.

Algoritmo 3.47 Algoritmo para pesquisa em uma *skip list* S.

Examinemos esse algoritmo mais de perto. Começamos o método SkipSearch com uma variável p marcando o item mais acima e à esquerda na *skip list* S. Ou seja, p marca a posição do item especial com chave $-\infty$ em S_h. Então realizamos os passos seguintes (veja a Figura 3.48):

1. Se S.below(p) é *null*, então a pesquisa termina – chegamos ao ***fundo*** e localizamos o maior item em S com chave menor ou igual a k. Caso contrário, descemos para o próximo nível na torre atual, fazendo $p \leftarrow S$.below(p).
2. Iniciando na posição p, movemos p para a frente até que ele esteja na posição mais à direita no nível atual tal que key(p) $\leq k$. Chamamos este passo de ***varredura***. Note que uma posição assim sempre existe, pois cada nível contém as chaves especiais $-\infty$ e $+\infty$. De fato, após realizarmos a varredura para este nível, p pode se manter onde estava inicialmente. Em qualquer caso, repetimos o passo anterior.

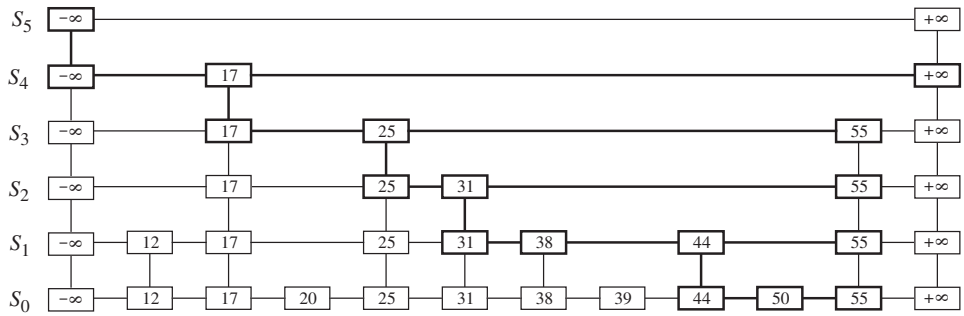


Figura 3.48 Exemplo de uma pesquisa em uma *skip list*. As posições visitadas e os *links* percorridos na procura (sem sucesso) pela chave 52 são mostrados com linhas grossas.

3.5.2 Operações de atualização

Dado o método SkipSearch, é simples implementar a operação findElement(k) – simplesmente fazemos $p \leftarrow$ SkipSearch (k) e testamos se key(p) = k. Quando isso ocorre, o tempo de execução esperado para o algoritmo SkipSearch é $O(\log n)$. Entretanto, postergamos esta análise até discutirmos os métodos de atualização para *skip lists*.

Inserção

O algoritmo de inserção para *skip lists* usa randomização para decidir quantas referências ao novo item (k, e) devem ser adicionadas à *skip list*. Começamos a inserção de um novo item (k, e) em uma *skip list* através de uma operação SkipSearch(k). Esta fornece a posição p do item no último nível com a maior chave menor ou igual a k (note que p pode ser o item especial com a chave $-\infty$). Inserimos então (k, e) nesta lista do último nível imediatamente após a posição p. Após inserir o novo item neste nível, "jogamos" uma moeda. Ou seja, usamos um método random() que retorna um número entre 0 e 1 e, se este número for menor do que 1/2, então consideramos que a moeda deu "coroa", e caso contrário, consideramos que a moeda deu "cara". Se ela der cara, paramos aqui. Se ela der coroa, por outro lado, subimos ao nível superior e inserimos (k, e) na posição adequada. Jogamos a moeda mais uma vez, e se ela der coroa, vamos ao próximo nível e repetimos. Assim, continuamos a inserir o novo item (k, e) em lis-

tas até que finalmente uma moeda dê cara. Em seguida ligamos todas as referências ao novo item (k, e) criadas neste processo para criar a ***torre*** para (k, e). Fornecemos o pseudocódigo para o algoritmo de inserção para uma *skip list S* no Trecho de Código 8.5 e exemplificamos esse algoritmo na Figura 8.11. Nosso algoritmo de inserção usa uma operação insertAfterAbove(p, q,(k, e)) que insere uma posição armazenando o item (k, e) após a posição p (no mesmo nível de p) e acima da posição q, retornando a posição r do novo item (e acertando as referências internas para que os métodos after, before, above e below funcionem corretamente para p, q e r).

Algoritmo SkipInsert(k, e)
  Entrada: item (k, e).
  Saída: nenhuma.
  $p \leftarrow$ SkipSearch(k)
  $q \leftarrow$ insertAfterAbove(p, **null**, (k, e)) {estamos no último nível}
  enquanto random() < 1/2 **faça**
    enquanto above(p) = **null faça**
      $p \leftarrow$ before(p) {varredura para trás}
    $p \leftarrow$ above(p) {subir ao nível superior}
    $q \leftarrow$ insertAfterAbove(p, q, (k, e)) {inserir novo item}

Algoritmo 3.49 Inserção em uma *skip list* assumindo que random() retorna um número aleatório entre 0 e 1 e que nunca inserimos acima do nível superior.

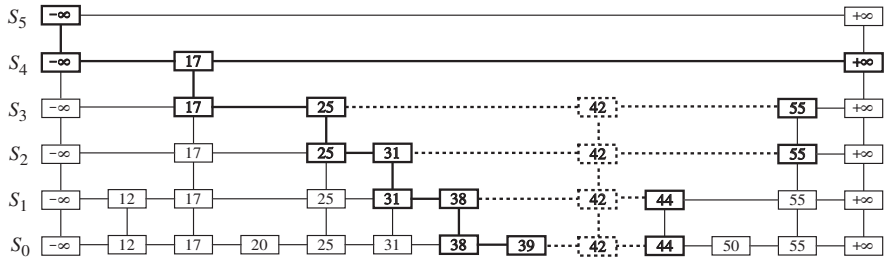


Figura 3.50 Inserção de um elemento com chave 42 na *skip list* da Figura 3.46. As posições visitadas e os *links* percorridos são desenhados com linhas grossas. As posições inseridas para guardar o novo item são desenhadas com linhas tracejadas.

Remoção

Assim como os algoritmos de pesquisa e inserção, o algoritmo de remoção para uma *skip list S* é bastante simples. De fato, ele é ainda mais simples do que o algoritmo de inserção. Para realizar uma operação removeElement(k), começamos realizando uma pesquisa pela chave k. Se uma posição p com a chave k não for encontrada, então retornamos o elemento NO_SUCH_KEY. Caso contrário, se a posição p com a chave k for encontrada (no último nível), então removemos todas as posições acima de p, que são facilmente acessadas usando a operação above para subir pela torre deste item em S começando na posição p. O algoritmo de remoção é ilustrado na Figura 3.51, e sua descrição detalhada é deixada como um exercício (R-3.19). Como mostraremos na próxima subseção, o tempo de execução esperado para remoção em uma *skip list* é $O(\log n)$.

Antes de descrever a análise, no entanto, existem alguns melhoramentos para a estrutura de dados *skip list* que devemos discutir. Primeiramente, não precisamos real-

mente armazenar referências para os itens da *skip list* acima do nível 0, pois tudo o que é necessário nestes níveis são referências para as chaves. Segundo: não precisamos realmente do método above; de fato, nem precisamos do método before. Podemos realizar a inserção e remoção de itens de cima para baixo, baseando-a em varreduras e economizando referências para os elementos anteriores e acima de um elemento. Exploraremos os detalhes desta otimização em um exercício (C-3.26).

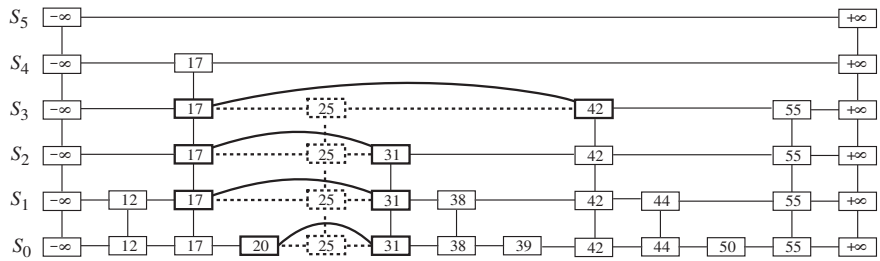


Figura 3.51 Remoção do item com chave 25 da *skip list* da Figura 8.11. As posições visitadas e os *links* percorridos antes da pesquisa inicial são desenhados com linhas grossas. As posições removidas são marcadas com linhas tracejadas.

Mantendo o nível superior

Uma *skip list S* deve manter uma referência ao elemento mais acima e à esquerda em *S* como uma variável instanciada, e devemos ter uma política para tratar qualquer inserção que queira continuar inserindo um item acima do nível superior de *S*. Existem duas opções possíveis neste caso – e as duas tem seus méritos.

Uma possibilidade é restringir o nível superior h para mantê-lo a algum valor fixo que é função de n, o número de itens correntes no dicionário (veremos, pela análise, que $h = \max\{10, 2\lceil \log n \rceil\}$ é uma boa escolha, e que $h = 3\lceil \log n \rceil$ é ainda mais seguro.) Implementar essa opção significa que devemos modificar o algoritmo de inserção para que ele termine quando atingirmos o nível mais alto (a não ser que $\lceil \log n \rceil < \lceil \log(n + 1) \rceil$ e, neste caso, podemos subir ainda mais um nível, pois o limite de altura está crescendo).

A outra possibilidade é deixar a inserção continuar inserindo o elemento enquanto a moeda lançada pelo gerador de números aleatórios der coroa. Como mostramos na análise das *skip lists*, a probabilidade de que uma inserção vá acima do nível $O(\log n)$ é muito baixa; portanto, esta escolha também deve funcionar.

Qualquer opção continua garantindo nossa habilidade de fazer a pesquisa por elementos, inserção e deleção em tempo esperado $O(\log n)$, o que mostraremos na próxima seção.

3.5.3 Análise probabilística das *skip lists*

Como mostramos anteriormente, *skip lists* fornecem uma implementação simples para um dicionário ordenado. Em termos do desempenho no pior caso, no entanto, *skip lists* não são uma estrutura de dados superior. De fato, se não impedirmos explicitamente que uma inserção ultrapasse o nível superior, então o algoritmo pode entrar em um laço praticamente infinito (ele não é, na verdade, um laço infinito, pois a probabilidade de uma moeda honesta dar sempre coroa é zero). Além disso, não podemos adicionar elementos infinitamente a uma lista sem, em algum momento, ficar sem memó-

ria disponível. Em qualquer caso, se terminarmos a inserção de um item no nível mais alto h, então o tempo de ***pior caso*** para as operações findElement, insertItem e removeElement em uma *skip list* S com n itens e altura h é de $O(n + h)$. Esse desempenho de pior caso ocorre quando a torre de cada item alcança o nível $h - 1$, onde h é a altura de S. No entanto, este evento tem uma probabilidade muito baixa e, julgando pelo pelo pior caso, podemos concluir que as *skip lists* são inferiores a outras implementações de dicionários discutidas anteriormente neste capítulo. Entretanto, essa não seria uma análise justa, pois o pior caso é exageradamente superestimado.

Como o passo de inserção envolve randomização, uma análise mais honesta das *skip lists* envolve um pouco de probabilidade. De início isso pode parecer uma grande empreitada, pois uma análise probabilística completa e minuciosa requereria matemática sofisticada (e, de fato, várias análises assim surgem na literatura de algoritmos e estruturas de dados). Felizmente, esse tipo de análise não é requerido para se compreender o comportamento assintótico das *skip lists*. A análise probabilística informal e intuitiva que faremos a seguir usa apenas conceitos básicos de teoria das probabilidades.

Limitando a altura em uma *skip list*

Começaremos determinando o valor esperado da altura h de S (supondo que não encerraremos a inserção em um determinado nível). A probabilidade de que um certo item seja armazenado em uma posição no nível i é igual à probabilidade de obter i caras consecutivas quando jogamos uma moeda, ou seja, a probabilidade é $1/2^i$. Portanto, a probabilidade P_i de que o nível i tenha ao menos um elemento é no máximo

$$P_i \leq \frac{n}{2^i},$$

pois a probabilidade de que qualquer um de n eventos diferentes ocorra é no máximo a soma das probabilidades de cada um deles.

A probabilidade de que a altura h de S seja maior do que i é igual à probabilidade de que o nível i tenha ao menos um item, ou seja, não é maior do que P_i. Isso significa que h é maior do que, por exemplo, $3 \log n$ com probabilidade de no máximo

$$P_{3\log n} \leq \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}.$$

Generalizando, dada uma constante $c>1$, h é maior do que $c \log n$ com probabilidade de no máximo $1/n^{c-1}$. Ou seja, a probabilidade de que h seja menor ou igual a $c \log n$ é de no mínimo $1 - 1/n^{c-1}$. Assim, com uma alta probabilidade, a altura h de S é $O(\log n)$.

Análise do tempo de pesquisa em uma *skip list*

Considere o tempo de execução de uma pesquisa em uma *skip list* S, lembrando que uma pesquisa envolve dois laços **enquanto** aninhados. O laço interno realiza a varredura em um nível de S enquanto a próxima chave não for maior do que a chave k sendo procurada e o laço externo desça para o nível inferior e repita a varredura. Como a altura h de S é $O(\log n)$ com grande probabilidade, o número de passos de descida nos níveis é $O(\log n)$, também com grande probabilidade.

Ainda temos de limitar o número de passos de varredura que fazemos. Seja n_i o número de chaves examinadas quando estamos fazendo uma varredura no nível i. Observe que, após a chave na posição inicial, cada chave adicional examinada em uma

varredura no nível i não pode pertencer ao nível $i + 1$, pois, se ela estivesse lá, nós a teríamos encontrado na varredura anterior. Assim, a probabilidade de que uma chave seja contada em n_i é 1/2. Portanto, o valor esperado de n_i é igual ao número esperado de vezes que devemos jogar uma moeda antes que ela dê cara. Vamos indicar esta quantidade por e. Teremos

$$e = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + e)$$

Logo, $e = 2$, e o total de tempo esperado que será gasto em varreduras em qualquer nível é $O(1)$. Como S tem $O(\log n)$ níveis com grande probabilidade, uma pesquisa em S toma um tempo esperado $O(\log n)$. Com uma análise similar, podemos mostrar que o tempo de execução esperado para inserção e deleção é $O(\log n)$.

Utilização de espaço em uma *skip list*

Finalmente, vamos examinar a exigência de espaço de uma *skip list* S. Como observamos acima, o número de itens esperado no nível i é $n/2^i$, significando que o número total de elementos esperados em S é

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n.$$

Portanto, a exigência de memória esperada para S é $O(n)$.

A Tabela 3.52 sumariza o desempenho de um dicionário realizado com uma *skip list*.

Operação	Tempo
keys, elements	$O(n)$
findElement, insertItem, removeElement	$O(\log n)$ (esperado)
findAllElements, removeAllElements	$O(\log n + s)$ (esperado)

Tabela 3.52 Desempenho de um dicionário realizado com uma *skip list*. Indicamos o número de itens no dicionário no momento da operação com n, e o tamanho do iterador retornado pelas operações findAllElements e removeAllElements com s. A exigência de memória esperada é $O(n)$.

3.6 Exemplo em Java: árvores AVL e árvores vermelho-pretas

Nesta seção, descrevemos uma classe genérica para árvores binárias, BinarySearchTree, e como ela pode ser estendida para produzir a implementação tanto de uma árvore AVL como de uma árvore vermelho-preta. A classe BinarySearchTree armazena pares chave-elemento da classe Item como os elementos armazenados nas posições (nodos) de sua árvore binária de suporte. O código de BinarySearchTree é apresentado nos Trechos de Código 3.54 a 3.56. Notamos que a árvore binária de suporte T é usada apenas através da interface BinaryTree, onde se supõe que BinaryTree inclua também os métodos expandExternal e removeAbovaExternal (veja a Seção 2.3.4). Assim, a classe BinarySearchTree tira vantagem da reutilização de código.

O método auxiliar findPosition, baseado no algoritmo TreeSearch, é invocado pelos métodos findElement, insertItem, e removeElement. A variável de instância actionPos armazena a posição onde a pesquisa, a inserção ou a remoção mais recente terminou. Esta variável de instância não é necessária para a implementação de uma árvore de pesquisa binária, mas é útil para as classes que irão estender BinarySearchTree (veja os Trechos de Código 3.58, 3.59, 3.62 e 3.63) para identificar a posição onde a pesquisa, a inserção ou a remoção anterior ocorreu. A posição actionPos é oferecida com a intenção de que seja usada após a execução dos métodos findElement, insertItem e removeElement.

```
public class Item {
    private Object key, elem;
    protected Item (Object k, Object e) {
      key = k;
      elem = e;
    {
    public Object key() {return key;}
    public Object element() {return elem;}
    public void setKey(Object k) {key = k;}
    public void setElement(Object e) {elem = e;}
}
```

Trecho de Código 3.53 Classe para os pares chave-elemento armazenados em um dicionário.

```
/** Implementação de um dicionário usando uma árvore de pesquisa binária */
public class BinarySearchTree implements Dictionary {
    Comparator C; // comparador
    BinaryTree T; // árvore binária
    protected Position actionPos; // posição de inserção ou pai da posição removida

    public BinarySearchTree(Comparator c){
      C = c;
      T = (BinaryTree) new NodeBinaryTree();
    }

    // métodos auxiliares
    /** Extrai a chave do item do nodo da árvore fornecido */
    protected Object key(Position position) {
      return ((Item) position. element()). key();
    }
    /** Extrai o elemento do item da árvore fornecido */
    protected Object element(Position position) {
      return ((Item) position.element()). element();
    }
    /** Verifica se a chave fornecida é válida */
    protected void checkKey(Object key) throws InvalidKeyException {
      if(!C.isComparable(key))
        throw new InvalidKeyException( "Key"+ key +" is not comparable");
}
```

Trecho de Código 3.54 Classe BinarySearchTree (continua no Trecho de Código 3.55).

```
/** Método auxiliar usado por removeElement */
protected void swap(Position swapPos, Position remPos){
    T.replaceElement(swapPos, remPos. element());
}
/** Método auxiliar usado por findElement, insertItem e removeElement */
protected Position findPosition (Object key, Position pos) {
    if (T.isExternal(pos))
      return pos; // chave não encontrada e nodo externo atingido retornado
    else {
      Object curKey = key(pos);
      if(C.isLessThan(key, curKey))
        return findPosition(key, T.leftChild(pos));
    else if(C.isGreaterThan(key, curKey)) // procura na subárvore esquerda
      return findPosition(key, T.rightChild(pos)); // procura na subárvore direita
    else
      return pos; // retorna o nodo interno onde a chave foi encontrada
    }
}

// métodos do TAD dicionário
public int size() {
    return (T.size() – 1) / 2;
}

public boolean isEmpty() {
    return T.size() == 1;
}

public Object findElement(Object key) throws InvalidKeyException {
    checkKey(key); // deve lançar InvalidKeyException
    Position curPos = findPosition(key, T.root());
    actionPos = curPos; // nodo onde a pesquisa se encerrou
    if (T.isInternal(curPos))
      return element(curPos);
    else
      return NO_SUCH_KEY;
}
```

Trecho de código 3.55 Classe BinarySearchTree, continuação do Trecho de Código 3.54 (continua no Trecho de Código 3.56).

```
public void insertItem (Object key, Object element)
    throws InvalidKeyException {
    checkKey(key); // pode disparar InvalidKeyException
    Position insPos = T.root();
    do {
      insPos = findPosition(key, insPos);
      if (T.isExternal(insPos))
        break;
      else // a chave já existe
        insPos = T.rightChild(insPos);
    } while (true);
    T.expandExternal (insPos);
    Item newItem = new Item(key, element);
    T.replaceElement(insPos, newItem);
    actionPos = insPos; // nodo onde o novo item foi inserido
}

public Object removeElement(Object key) throws InvalidKeyException {
    Object toReturn;
    checkKey(key); // pode disparar InvalidKeyException
    Position remPos = findPosition(key, T.root());
    if(T.isExternal(remPos)) {
      actionPos = remPos; // nodo onde a pesquisa terminou sem sucesso
      return NO_SUCH_KEY;
    }
    else{
      toReturn = element(remPos); // elemento a ser retornado
      if (T.isExternal(T.leftChild(remPos)))
        remPos = T.leftChild(remPos);
      else if (T. isExternal(T. rightChild (remPos)))
        remPos = T.rightChild(remPos);
      else { // chave no nodo com filho interno
        Position swapPos = remPos; // encontra nodo para trocar itens
        remPos = T.rightChild(swapPos);
        do
          remPos = T.leftChild(remPos);
        while (T.isInternal(remPos));
        swap(swapPos, T.parent(remPos));
      }
      actionPos = T.sibling(remPos); // irmão da folha a ser removida
      T. removeAboveExternal (remPos);
      return toReturn;
      }
    }
}
```

Trecho de Código 3.56 Classe BinarySearchTree, continuação do Trecho de Código 3.55.

3.6.1 Implementação Java de árvores AVL

Vamos agora nos concentrar em detalhes de implementação e analisar o uso de uma árvore AVL T com n nodos internos para implementar um dicionário ordenado de n itens. Os algoritmos de inserção e remoção para T requerem que sejamos capazes de executar a reestruturação trinodo e determinar a diferença entre as altura de dois nodos irmãos. Falando em restruturações, podemos estender a coleção de operações de um TAD árvore binária adicionando o método restructure. É fácil perceber que a operação restructure pode ser executada em tempo $O(1)$ se T for implementado usando uma estrutura encadeada (Seção 2.3.4), o que não será o caso com um vetor (Seção 2.3.4). Desta forma, vamos preferir uma estrutura encadeada para representar uma árvore AVL.

Em relação à informação sobre altura, podemos armazenar explicitamente a informação de altura de cada nodo interno v no próprio nodo. Outra opção é armazenarmos o ***fator de balanceamento*** de v em v, que é definido como a altura do filho da esquerda de v menos a altura do filho da direita de v. Em conseqüência, o fator de balanceamento de v será sempre –1, 0 ou 1, exceto durante uma inserção ou remoção, quando poderá ser ***temporariamente*** igual a –2 ou +2. Durante a execução de uma inserção ou remoção as alturas e fatores de balanceamento de $O(\log n)$ nodos são afetados e podem ser reparados em tempo $O(\log n)$.

Nos Trechos de Código 3.57 a 3.59, apresentamos uma implementação Java de um dicionário usando uma árvore AVL. A classe AVLItem, apresentada no Trecho de Código 3.57, estende a classe Item usada para representar um item chave-elemento de uma árvore de pesquisa binária. Ela define uma variável de instância adicional height, que representa a altura do nodo. A classe AVLTree, apresentada por completo nos Trechos de Código 3.58 e 3.59, estende BinarySearchTree (Trechos de Código 3.54 a 3.56). O construtor de AVLTree executa o construtor da superclasse primeiramente e então atribui uma RestructurableNodeBinaryTree para T, que é uma classe que implementa o TAD árvore binária e suporta adicionalmente o método restructure para executar reestruturações trinodo. A classe AVLTree herda os métodos size, isEmpty, findElement, findAllElements e removeAllElements de sua superclasse BinarySearchTree, mas sobrecarrega os métodos insertItem e removeElement.

O método insertItem (Trecho de Código 3.59) inicia chamando o método insertItem da superclasse, que insere o novo item e atribui a posição de inserção (nodo que armazena a chave 54 na Figura 3.12) para a variável de instância actionPos. O método auxiliar rebalance é então usado para percorrer o caminho desde a posição de inserção até a raiz. Este caminhamento atualiza a altura de todos os nodos visitados e executa uma reestruturação trinodo, se necessário. Da mesma forma, o método removeElement (Trecho de Código 3.59) inicia chamando o método da superclasse removeElement, que executa a remoção do item e atribui a posição que substitui o eliminado para a variável de instância actionPos. O método auxiliar rebalance é então usado para percorrer o caminho desde a posição removida até a raiz.

```
public class AVLItem extends Item {
    int height;
    AVLItem(Object k, Object e, int h) {
      super(k, e);
      height = h;
    }
    public int height() {return height;}
    public int setHeight(int h) {
      int oldHeight = height;
      height = h;
      return oldHeight;
    }
}
```

Trecho de Código 3.57 Classe que implementa um nodo de uma árvore AVL. A altura do nodo na árvore é armazenada em uma variável de instância.

```
/** Implementação de um dicionário usando uma árvore AVL */
public class AVLTree extends BinarySearchTree implements Dictionary {
    public AVLTree(Comparator c) {
      super(c);
      T = new RestructurableNodeBinaryTree();
    }
    private int height(Position p) {
      if (T.isExternal (p))
        return 0;
      else
        return ((AVLItem) p.element()). height();
    }
    private void setHeight(Position p) { // chamada apenas se p for interno
      ((AVLItem) p.element()). setHeight(1 + Math.max(height(T.leftChild(p)),
                                                      height(T.rightChild(p))));
    }
    private boolean isBalanced (Position p) {
      // testa se o nodo p tem fator de balanceamento entre –1 e 1
      int bf = height(T.leftChild(p)) – height(T.rightChild(p));
      return ((–1 <= bf) && (bf <= 1));
    }
    private Position tallerChild(Position p) {
      // retorna o filho de p cuja altura não é menor que a do outro filho
      if(height(T.leftChild(p)) >= height(T.rightChild(p)))
        return T.leftChild(p);
      else
        return T.rightChild(p);
    }
```

Trecho de Código 3.58 Métodos construtor e auxiliar da classe AVLTree.

```
/**
  * Método auxiliar chamado por insertItem e removeElement.
  * Percorre o caminho de T partindo do nodo dado até a raiz. Para
  * cada nodo zPos encontrado, recalcula a altura de zPos e executa
  * uma reestruturação trinodo se zPos estiver desbalanceado.
*/
private void rebalance(Position zPos) {
      while (!T.isRoot(zPos)) {
        zPos = T.parent(zPos);
        setHeight(zPos);
        if (!isBalanced(zPos)) {
          // executa a reestruturação trinodo
          Position xPos = tallerChild(tallerChild(zPos));
          zPos = ((RestructurableNodeBinaryTree) T). restructure(xPos);
          setHeight(T.leftChild(zPos));
          setHeight(T.rightChild(zPos));
          setHeight(zPos);
        }
      }
}

// métodos do TAD dicionário

/** Sobrecarrega o método correspondente da classe pai */
public void insertItem (Object key, Object element)
        throws InvalidKeyException {
      super.insertItem (key, element); // pode disparar uma InvalidKeyException
      Position zPos = actionPos; // inicia na posição de inserção
      T.replaceElement(zPos, new AVLItem(key, element, 1));
      rebalance(zPos);
}

/** Sobrecarrega o método correspondente da classe pai */
public Object removeElement(Object key)
        throws InvalidKeyException {
      Object toReturn = super.removeElement(key);
                                    // pode disparar uma InvalidKeyException
if(toReturn != NO_SUCH_KEY) {
        Position zPos = actionPos; // inicia na posição de remoção
        rebalance(zPos);
      }
      return toReturn;
}
```

Trecho de Código 3.59 Método auxiliar rebalance e métodos de dicionário e removeElement da classe AVLTree.

3.6.2 Implementação Java de árvores vermelho-pretas

Nos Trechos de Código 3.60 a 3.63, apresentamos trechos da implementação Java de um dicionário organizado usando uma árvore vermelho-preta. A classe RBTItem, apresentada no Trecho de Código 3.60, estende a classe Item, usada para representar um elemento chave de uma árvore de pesquisa binária. Ela define uma variável de instância adicional, isRed, que representa a cor do nodo, bem como métodos para alterar e retornar a mesma.

```
public class RBTItem extends Item {
    private boolean isRed;
    public RBTItem(Object k, Object elem, boolean color) {
      super(k, elem);
      isRed = color;
    }
    public boolean isRed() {return isRed;}
    public void makeRed() {isRed = true;}
    public void makeBlack() {isRed = false;}
    public void setColor(boolean color) {fisRed = color;}
}
```

Trecho de Código 3.60 Classe que implementa um nodo de uma árvore vermelho-preta.

```
/** Implementação de um dicionário usando uma árvore vermelho-preta */
public class RBTree extends BinarySearchTree implements Dictionary {
    static boolean Red = true;
    static boolean Black = false;

    public RBTree(Comparator C) {
      super(C);
      T = new RestructurableNodeBinaryTree();
}
```

Trecho de Código 3.61 Variáveis de instância e construtor da classe RBTree.

A classe RBTree, apresentada parcialmente nos trechos de código 3.61 a 3.63, estende BinarySearchTree (trechos de código 3.54 a 3.56). Como na classe AVLTree, o construtor de RBTree executa primeiramente o construtor da superclasse e então atribui a T uma RestructurableNodeBinaryTree, que é uma classe que implementa o TAD árvore binária e, além disso, suporta o método restructure para executar reestruturações trinodo (rotações). A classe RBTree herda os métodos size, isEmpty, findElement, findAllElements e removeAllElements de BinarySearchTree, mas sobrecarrega os métodos insertItem e removeElement. Diversos métodos auxiliares da classe RBTree não são mostrados.

Os métodos insertItem (Trecho de Código 3.62) e removeElement (Trecho de Código 3.63) chamam primeiramente os métodos correspondentes da superclasse, e então balanceiam a árvore chamando métodos auxiliares para executarem rotações ao longo do caminho partindo da posição de inserção (dada pela variável de instância actionPos herdada da superclasse) até a raiz.

```
public void insertItem (Object key, Object element)
      throws InvalidKeyIException {
    super.insertItem (key, element); // pode disparar uma InvalidKeyException
    Position posZ = actionPos; // inicia na posição de inserção
    T.replaceElement(posZ, new RBTItem(key, element, Red));
    if (T.isRoot(posZ))
      setBlack(posZ);
    else
      remedyDoubleRed (posZ);
}

protected void remedyDoubleRed (Position posZ) {
    Position posV = T.parent(posZ);
    if (T.isRoot(posV))
      return;
    if (!isPosRed(posV))
      return;
    // temos um vermelho duplo: posZ e posV
    if (!isPosRed(T.sibling(posV))) { // Caso 1: reestruturação trinodo
      posV = ((RestructurableNodeBinaryTree) T).restructure(posZ);
      setBlack(posV);
      setRed(T.leftChild(posV));
      setRed(T.rightChild(posV));
    }
    else  { // Caso 2: troca de cores
      setBlack(posV);
      setBlack(T.sibling(posV));
      Position posU = T.parent(posV);
      if (T.isRoot(posU))
        return;
      setRed(posU);
      remedyDoubleRed (posU);
    }
}
```

Trecho de Código 3.62 Métodos de dicionário insertItem e método auxiliar remedyDoubleRed da classe RBTree.

```
public Object removeElement(Object key) throws InvalidKeyException {
    Object toReturn = super.removeElement (key);
    Position posR = actionPos;
    if (toReturn ! = NO_SUCH_KEY) {
      if (wasParentRed(posR) || T.isRoot(posR) || isPosRed(posR))
        setBlack(posR);
      else
        remedyDoubleBlack(posR);
    }
    return toReturn;
}
protected void remedyDoubleBlack(Position posR) {
    Position posX, posY, posZ;
    boolean oldColor;
    posX = T.parent(posR);
    posY = T.sibling(posR);
    if (!isPosRed(posY)) {
    posZ = redChild(posY);
    if (hasRedChild(posY))  { // Caso 1: reestruturação trinodo
      oldColor = isPosRed(posX);
      posZ = ((RestructurableNodeBinaryTree) T).restructure(posZ);
      setColor(posZ, oldColor);
      setBlack(posR);
      setBlack(T.leftChild(posZ));
      setBlack(T.rightChild(posZ));
      return;
    }
    setBlack(posR);
    setRed(posY);
    if (!isPosRed(posX)) // Caso 2: recolorindo
      if (!T.isRoot(posX))
        remedyDoubleBlack(posX);
      return;
    }
    setBlack(posX);
    return;
} // Caso 3: ajuste
if (posY == T.rightChild(posX))
    posZ = T.rightChild(posY);
else
    posZ = T.leftChild(posY);
    ((RestructurableNodeBinaryTree)T).restructure(posZ);
    setBlack(posY);
    setRed (posX);
    remedyDoubleBlack (posR);
}
```

Trecho de Código 3.63 Método removeElement e seu método auxiliar.

3.7 Exercícios

Reforço

R-3.1 Insira, em uma árvore binária de pesquisa inicialmente vazia, itens com as seguintes chaves (nesta ordem): 30, 40, 24, 58, 48, 26, 11, 13. Desenhe a árvore após cada inserção.

R-3.2 Um certo professor Amongus afirma que não importa a ordem pela qual um conjunto fixo de elementos é inserido em uma árvore binária de pesquisa, sempre resulta a mesma árvore. Apresente um pequeno exemplo que prove que o professor Amongus está errado.

R-3.3 O professor Amongus afirma que ele tem uma "correção" à sua afirmativa do exercício anterior, ou seja, que a ordem pela qual um conjunto fixo de elementos é inserido em uma árvore AVL não importa – irá resultar sempre na mesma árvore AVL. Apresente um pequeno exemplo que prove que o professor Amongus ainda está errado.

R-3.4 A rotação feita na Figura 3.12 é simples ou dupla? E quanto à rotação da Figura 3.15?

R-3.5 Desenhe a árvore AVL resultante da inserção de um item com chave 52 na árvore AVL da Figura 3.15b.

R-3.6 Desenhe a árvore AVL resultante da remoção do item com chave 62 da árvore AVL da Figura 3.15b.

R-3.7 Explique por que executar uma rotação em uma árvore binária com n nodos implementada usando uma seqüência consome tempo $\Omega(n)$.

R-3.8 A árvore de pesquisa genérica da Figura 3.17a corresponde a uma árvore (2,4)? Justifique sua resposta.

R-3.9 Uma forma alternativa de executar uma divisão em um nodo v em uma árvore (2,4) é partir v em v' e v'', com v' sendo um 2-nodo e v'' sendo um 3-nodo. Qual das chaves k_1, k_2, k_3 ou k_4 armazenamos no pai de v, neste caso? Por quê?

R-3.10 O professor Amongus afirma que uma árvore (2,4) que armazena um conjunto de itens sempre terá a mesma estrutura, não importando a ordem que os itens sejam inseridos. Mostre que o professor Amongus está errado.

R-3.11 Considere a seguinte seqüência de chaves:

$$(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1)$$

Considere a inserção de itens com essas chaves, nessa ordem, em:

a. uma árvore (2, 4) T' inicialmente vazia;
b. uma árvore vermelha e preta T'' inicialmente vazia.

Desenhe T' e T'' após cada inserção.

R-3.12 Desenhe quatro árvores vermelho-pretas diferentes que correspondam à mesma árvore (2, 4) usando as regras de correspondência descritas neste capítulo.

R-3.13 Desenhe um exemplo de árvore vermelho-preta que não seja uma árvore AVL. Sua árvore deve ter ao menos seis nodos, mas não pode ter mais de 16.

R-3.14 Para cada uma das seguintes afirmações sobre árvores vermelho-preta determine quais são verdadeiras e quais são falsas. Se você achar que é verdadeira, justifique. Se você achar que é falsa, dê um contra-exemplo.

a. Uma subárvore de uma árvore vermelho-preta é também uma árvore vermelho-preta.
b. O irmão de um nodo externo ou é externo ou é vermelho.
c. Dada uma árvore vermelho-preta T, existe apenas uma única árvore (2, 4) T' associada com T.
d. Dada uma árvore (2, 4) T, existe uma única árvore vermelho-preta T' associada com T.

R-3.15 Execute a seguinte seqüência de operações em uma *splay tree* inicialmente vazia e desenhe a árvore após cada operação.

a. Insira as chaves 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, nessa ordem.
b. Procure por chaves 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, nessa ordem.
c. Delete as chaves 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, nessa ordem.

R-3.16 Como se parece uma *splay tree* se os seus itens forem acessados na ordem crescente de suas chaves?

R-3.17 Quantas operações de reestruturação trinodo são necessárias para executar as atualizações zigue-zigue, zigue-zague e zigue em *splay trees*? Use figuras para explicar seu cálculo.

R-3.18 Desenhe um exemplo de *skip list* resultante da execução da seguinte seqüência de operações na *skip list* da Figura 3.51: removeElement(38), insertItem(48, x), insertItem(24, y), removeElement(55). Suponha que a moeda vire para a primeira inserção habilitando duas caras seguidas por coroas e para a segunda inserção habilitando três caras seguidas por coroas.

R-3.19 Escreva uma descrição em pseudocódigo da operação do dicionário removeElement, supondo que o dicionário é implementado por uma estrutura *skip list*.

Criatividade

C-3.1 Suponha dois dicionários ordenados S e T, cada um com n itens, e que S e T são implementados usando seqüências ordenadas baseadas em arranjo. Descreva um algoritmo que execute em tempo $O(\log n)$ para determinar a k-ésima menor chave na união das chaves de S e T (assuma que não existem chaves duplicadas).

C-3.2 Projete um algoritmo que execute o procedimento findAllElements(k) sobre um dicionário ordenado implementado usando um arranjo ordenado e mostre que ele executa em tempo $O(\log n + s)$, onde n corresponde a quantidade de elementos no dicionário e s é o número de itens retornados.

C-3.3 Projete um algoritmo que execute o procedimento findAllElements(k) sobre um dicionário ordenado implementado usando uma árvore de pesquisa binária T, e mostre que ele executa em tempo $O(h + s)$, onde h corresponde à altura de T no dicionário e s é o número de itens retornados.

C-3.4 Descreva como executar a operação removeAllElements(k) em um dicionário ordenado implementado usando uma árvore de pesquisa binária T e mostre que ele executa em tempo $O(h + s)$, onde h é a altura da árvore de pesquisa T, e s é o tamanho do iterador retornado.

C-3.5 Desenhe um exemplo de árvore AVL cuja operação removeElement requeira $\Theta(\log n)$ reestruturações (ou rotações) trinodo da folha até a raiz de maneira a restaurar a propriedade de altura/balanceamento. (Use triângulos para representar as subárvores que não são afetadas por esta operação.)

C-3.6 Mostre como executar a operação removeAllElements(k) em um dicionário implementado com uma árvore AVL em tempo $O(s \log n)$, onde n é o número de elementos no dicionário no momento em que a operação é executada, e s é o tamanho do iterador retornado pela operação.

C-3.7 Se mantivermos uma referência para a posição do nodo mais interno à esquerda de uma árvore AVL, então a operação first pode ser executada em tempo $O(1)$. Descreva como a implementação dos outros métodos do dicionário necessitam ser modificadas de maneira a manter uma referência para a posição mais à esquerda.

C-3.8 Mostre que qualquer árvore binária com n nodos pode ser convertida em qualquer outra árvore binária de n nodos usando $O(n)$ rotações.
Dica*:** mostre que $O(n)$ rotações são suficientes para converter qualquer árvore binária em uma ***cadeia esquerda onde cada nodo interno tenha um filho externo à direita.

C-3.9 Mostre que os nodos que se tornam desbalanceados em uma árvore AVL após a execução da operação expandExternal, em conseqüência de uma operação insertItem, podem não ser consecutivos no caminho do nodo recém-inserido até a raiz.

C-3.10 Faça D ser um dicionário ordenado com n itens implementado usando uma árvore AVL. Mostre como implementar a operação a seguir sobre D em tempo $O(\log n + s)$, onde s é o tamanho do iterador retornado:

findAllInRange(k_1, k_2): retorna um iterador de todos os elementos de D com chave k onde $k_1 \leq k \leq k_2$.

C-3.11 Seja D um dicionário ordenado com n itens implementado usando uma árvore AVL. Mostre como implementar o seguinte método para D em tempo $O(\log n)$:

countAllInRange(k_1, k_2): calcula e retorna o número de itens em D com chave k de maneira que $k_1 \leq k \leq k_2$.

Observe que esse método retorna um único inteiro.
***Dica*:** você vai precisar estender a estrutura de dados de uma árvore AVL, acrescentando um novo campo para cada nodo interno e formas de manter este campo durante as atualizações.

C-3.12 Mostre que pelo menos um nodo de uma árvore AVL se torna desbalanceado, após a operação removeAboveExternal ser executada, em conseqüencia da execução de uma operação de dicionário removeElement.

C-3.13 Mostre que no mínimo uma operação de reestruturação trinodo (que corresponde a uma rotação simples ou dupla) é necessária para restaurar o balanceamento após qualquer inserção em uma árvore AVL.

C-3.14 Sejam T e U árvores (2,4) que armazenam n e m itens, respectivamente, tais que todos os itens de T têm menos chaves que todos os itens de U. Descreva um método que executa em $O(\text{lon } n + \log m)$ para unir T e U em uma única árvore que armazena todos os itens de T e U (destruindo as versões antigas de T e U).

C-3.15 Repita o problema anterior para árvores vermelho-pretas T e U.

C-3.16 Justifique o Teorema 3.3.

C-3.17 O marcador booleano usado para indicar nodos em uma árvore vermelho-preta como sendo "vermelho" ou "preto" não é absolutamente necessário. Descreva um esquema para implementar uma árvore vermelho-preta sem acrescentar espaço extra para nodos padrão de pesquisa binária. Como o seu esquema afeta os tempos de execução de pesquisa e atualização de uma árvore vermelho-preta?

C-3.18 Seja T uma árvore vermelho-preta que armazena n itens, e seja k a chave de um item em T. Mostre como construir em tempo $O(\log n)$, a partir de T, duas árvores vermelho-pretas T' e T'' de maneira que T' contenha todas as chaves de T menores do que k, e T'' contenha todas as chaves de T maiores do que k. Essa operação destrói T.

C-3.19 O TAD ***mergeable heap*** consiste nas operações insert(k,x), remove(k), unionWith(h) e minElement(), onde a operação unionWith(h) executa a união do *mergeable-heap h* com o atual, destruindo as versões antigas de ambos. Descreva uma implementação concreta para o TAD *mergeable heap* que obtenha performance $O(\log n)$ para todas as suas operações. Para simplificar, você pode assumir que todas as chaves existentes em *mergeable heap* sejam distintas, embora isso não seja absolutamente necessário.

C-3.20 Considere uma variação das *splay trees* chamada de *meias-splay-trees*, onde a aplicação do *splaying* sobre um nodo em uma profundidade d se encerra tão-logo o nodo atinja a profundidade $\lfloor d/2 \rfloor$. Execute uma análise amortizada das *meias-splay-trees*.

C-3.21 O passo padrão da operação de *splaying* requer duas etapas: um passo de descida para encontrar o nodo x sobre o qual aplicar o *splaying* e um de subida aplicando o *splaying* sobre o nodo x. Descreva um método que aplique o *splaying* e pesquise em um único passo. Cada passo intermediário requer agora que você considere os próximos dois nodos no caminho em direção a x, com um possível zigue sendo executado no fim. Descreva os detalhes da execução de cada passo intermediário zigue-zigue, zigue-zague e zigue.

C-3.22 Descreva uma seqüência de acessos a uma *splay tree* de n nodos, onde n é ímpar, que resulte em T consistindo em uma cadeia simples de nodos inter-

nos com nodos externos como filhos, de forma que o caminho interno de descida em T alterne entre filhos da direita e da esquerda.

C-3.23 Justifique o Teorema 3.11. Uma forma de demonstrar essa justificativa é observar que podemos redefinir o "tamanho" de um nodo como sendo a soma das freqüências de acesso a seus filhos e mostrar que toda justificativa do Teorema 3.9 ainda vale.

C-3.24 Suponha que temos uma seqüência ordenada S de itens $(x_0, x_1,..., x_{n-1})$ de maneira que, para cada item x_i de S, é fornecido um peso inteiro e positivo a_i. Faça A denotar o peso total de todos os elementos de S. Desenvolva um algoritmo que consuma $O(n \log n)$ e que construa uma árvore de pesquisa T para S de maneira que a profundidade de cada item a_i seja $O(\log A/a_i)$.

Dica: Encontre o item x_j com maior j tal que $\sum_{i=0}^{j-1} a_i < A/2$. Considere colocar este item na raiz e aplicar recursividade nas duas subseqüências induzidas.

C-3.25 Projete um algoritmo de tempo linear para o problema anterior.

C-3.26 Mostre que os métodos above(p) e before(p) não são na verdade necessários para uma implementação eficiente de um dicionário usando uma *skip list*. Isto é, podemos implementar a inserção e remoção de itens em uma *skip list* usando somente uma abordagem *top-down* para frente, sem nunca usar os métodos above ou before.

C-3.27 Descreva como implementar o método localizador before(ℓ) bem como o método localizador closestBefore(k) em um dicionário implementado usando uma seqüência ordenada. Faça o mesmo para uma implementação que use uma seqüência não-ordenada. Quais os tempos de execução desses métodos?

C-3.28 Repita o exercício anterior usando uma *skip list*. Quais os tempos esperados de execução para os dois métodos de localização em sua implementação?

C-3.29 Suponha que cada linha de um arranjo A de tamanho $n \times n$ seja composta por "zeros" e "uns" de maneira que, em qualquer linha de A, todos os "uns" estejam localizados antes dos "zeros" daquela linha. Assumindo que A esteja na memória, descreva um método que execute em tempo $O(n \log n)$ (não em tempo $O(n^2)$!) para contar o número de "uns" em A.

C-3.30 Descreva uma estrutura eficiente de dicionário ordenado para armazenar n elementos que tenham um conjunto associado de chaves $k < n$ originárias de um conjunto de ordem total, isto é, o conjunto de chaves é menor que o número de elementos. Sua estrutura deve executar todas as operações de um dicionário ordenado em tempo $O(\log k + s)$, onde s é o número de elementos retornados.

Projetos

P-3.1 Implemente os métodos de um TAD dicionário ordenado, usando uma árvore AVL, uma *skip list* ou uma árvore vermelho-preta.

P-3.2 Forneça uma animação gráfica ds operações de uma *skip list*. Visualize como os itens movem-se na lista durante as inserções e como são desconectados durante as remoções.

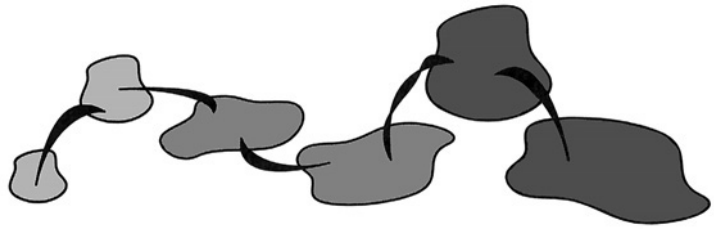
Notas

Algumas das estruturas de dados discutidas neste capítulo são descritas em detalhes por Knuth em [119] e por Mehlhorn em [148]. Árvores AVL são atribuídas a Adel'son-Vel'skii e Landis [2]. Análises de altura média para árvores de pesquisa binária podem ser encontradas nos livros de Aho, Hopcroft e Ullman [8] e Cormen, Leiserson e Rivest [55]. O livro de bolso de Gonnet e Baeza-Yates [81] contém uma boa quantidade de comparações experimentais e teóricas entre implementações de dicionários. Aho, Hopcroft e Ullman [7], discutem árvores (2, 3), que são similares a árvores (2, 4). Árvores vermelho-pretas são definidas por Bayer [23], e são também discutidas por Guibas e Sedgewick [91]. *Splay trees* foram inventadas por Sleator e Tarjan [189] (veja também [200]). Leituras adicionais podem ser encontradas nos livros de Mehlhorn [148] e Tarjan [200], e o capítulo por Mehlhorn e Tsakalidis [152]. Knuth [119] é uma leitura adicional excelente que inclui abordagens mais recentes de árvores balanceadas. O Exercício C-3.25 foi inspirado em um artigo de Mehlhorn [147]. *Skip lists* foram introduzidas por Pugh [170]. Nossa análise de *skip lists* é uma simplificação de uma apresentação encontrada no livro de Motwani e Raghavan [157]. Ao leitor interessado em outras construções probabilísticas para suportar o TDA dicionário, é indicado o texto de Motwani e Raghavan [157]. Para uma análise mais profunda sobre *skip lists*, são indicados ao leitor artigos a respeito [115, 163, 167].

Capítulo

Ordenação, conjuntos e seleção

4



4.1 *Merge-sort*. 223

4.1.1 Divisão e conquista 223

4.1.2 O *merge-sort* e suas relações de recorrência 228

4.2 O TAD conjunto. 229

4.2.1 Uma implementação simples para conjunto 230

4.2.2 Partições com operações *union* e *find* 231

4.2.3 Uma implementação de partição usando árvores 232

4.3 *Quick-sort* 239

4.3.1 *Quick-sort* randomizado 242

4.4 Limite inferior em ordenação baseada em comparação 243

4.5 *Bucket-sort* e *radix-sort* 245

4.5.1 *Bucket-sort* 245

4.5.2 *Radix-sort* 246

4.6 Comparação dos algoritmos de ordenação 248

4.7 Seleção 249

4.7.1 Poda e busca 249

4.7.2 *Quick-select* randomizado 250

4.7.3 Analisando o *quick-select* randomizado 250

4.8 Exemplo em Java: *in-place quick-sort* 252

4.9 Exercícios 255

A Segunda Lei da Termodinâmica sugere que a natureza tende para a desordem. Os seres humanos, por outro lado, preferem a ordem. De fato, existem diversas vantagens em se manterem os dados em ordem. Por exemplo, o algoritmo de pesquisa binária funciona corretamente apenas com vetores ordenados. Uma vez que os computadores foram criados para ser ferramentas para os homens, dedicamos este capítulo ao estudo de algoritmos de ordenação e suas aplicações. Lembramos que o problema da ordenação é definido como segue. Seja S uma seqüência de n elementos que podem ser comparados entre si de acordo com uma relação total de ordem, ou seja, é sempre possível comparar dois elementos de S para verificar qual é maior ou menor ou se os dois são iguais. Queremos reorganizar S de maneira que os elementos apareçam em ordem crescente (ou não decrescente se houver elementos iguais em S).

Já apresentamos diversos algoritmos de ordenação nos capítulos anteriores. Em especial, na Seção 2.4.2, mostramos um esquema de ordenação simples, chamado PriorityQueueSort que consiste na inserção de elementos em uma fila de prioridades e sua extração em ordem não decrescente através de uma série de operações removeMin. Se a fila de prioridades é implementada usando uma seqüência, então o algoritmo executa em tempo $O(n^2)$ e corresponde ao método de ordenação conhecido tanto como *insertion-sort* ou *selection-sort*, dependendo se a seqüência mantida na fila de prioridades é conservada ordenada ou não. Se, ao contrário, a fila de prioridades é implementada em termos de um *heap* (Seção 2.4.3) então o algoritmo executa em tempo $O(n \log n)$ e corresponde ao método de ordenação conhecido como *heap-sort*.

Neste capítulo, apresentamos outros algoritmos de ordenação, juntamente com os padrões de projeto em que se baseiam. Dois desses algoritmos, chamados de ***merge-sort*** e ***quick-sort***, baseiam-se no padrão de projeto "divisão e conquista", que é extensamente aplicável a outros problemas. Outros dois algoritmos, ***bucket-sort*** e ***radix-sort***, baseiam-se, por sua vez, na abordagem de arranjo de *bucket* utilizada com *hashing*. Introduzimos também o tipo abstrato de dados ***conjunto*** e mostramos como a técnica de junção, usada no algoritmo *merge-sort*, pode ser usada na implementação destes métodos. O tipo abstrato de dados conjunto tem um subtipo importante conhecido como ***partição***, que suporta os métodos primários union e find e tem uma implementação muito rápida. De fato, mostraremos que uma seqüência de n operações union e find pode ser implementada no tempo $O(n \log^* n)$, onde $\log^* n$ é o número de vezes que a função logarítmica pode ser aplicada, iniciando em n, antes de alcançar 1. Esta análise provê um exemplo incomum de análise amortizada.

Discutiremos também neste capítulo a prova *lower bound* para o problema de ordenação, mostrando que qualquer aproximação baseada em comparação precisa executar no mínimo $\Omega(n \log n)$ operações para ordenar n números. Em alguns casos, entretanto, não estamos interessados em ordenar um conjunto inteiro, mas gostaríamos de selecionar o k-ésimo menor elemento no conjunto. Mostraremos que este problema de ***seleção*** pode, de fato, ser resolvido mais rapidamente que o problema de ordenação. O exemplo de implementação em Java que conclui o capítulo é o *in-place quick-sort*.

Neste capítulo, assumiremos que é definida uma relação total de ordem sobre os elementos a serem ordenados. Se esta relação é induzida por um comparador (Seção 2.4.1), assumimos que o teste de comparação leva tempo $O(1)$.

4.1 *Merge-sort*

Nesta seção, mostramos uma técnica de ordenação chamada ***merge-sort***, que pode ser descrita de uma forma simples e compacta, usando recursão.

4.1.1 Divisão e conquista

O *merge-sort* é baseado em um algoritmo de padrão de projeto chamado de ***divisão e conquista*** (*divide-and-conquer*). O paradigma de divisão e conquista pode ser descrito, de maneira geral, como sendo composto de três fases:

1. ***Divisão:*** se o tamanho da entrada for menor que um certo limiar (digamos, um ou dois elementos), resolvemos o problema usando um método direto e retornamos a solução obtida. Em qualquer outro caso, dividimos os dados de entrada em dois ou mais conjuntos separados.
2. ***Recursão:*** solucionamos os problemas associados aos subconjuntos recursivamente.
3. ***Conquista:*** obtemos as soluções dos subproblemas e juntamos as mesmas em uma única solução para o problema original.

Merge-sort aplica a técnica divisão e conquista para o problema de ordenação.

Uso de divisão e conquista para ordenação

Lembramos que, no problema da ordenação, temos uma coleção de n objetos, tipicamente armazenados em uma lista, vetor, arranjo ou seqüência, com algum comparador que define uma relação total de ordem nesses objetos, e devemos gerar uma representação ordenada destes. Para salvaguarda da generalidade, enfocamos a versão do problema de ordem que toma uma seqüência S por entrada e retorna S ordenado. Especializações para outras estruturas lineares tais como listas, vetores ou arranjos são diretas e incluídas como exercício (R-4.3 e R-4.13). Para o problema de ordenação de uma seqüência de n elementos, os três passos de divisão e conquista são os seguintes:

1. ***Divisão****:* se S tiver zero ou um elemento, retorne S imediatamente; já está ordenado. Em qualquer outro caso (S tem pelo menos dois elementos), remova todos os elementos de S e coloque-os em duas seqüências, S_1 e S_2, cada uma contendo aproximadamente a metade dos elementos de S, ou seja, S_1 contém os primeiros $\lceil n/2 \rceil$ elementos de S, e S_2 contém os restantes $\lfloor n/2 \rfloor$ elementos.
2. ***Recursão****:* recursivamente, ordene as seqüências S_1 e S_2.
3. ***Conquista****:* coloque os elementos de volta em S, unindo as seqüências S_1 e S_2 em uma seqüência ordenada.

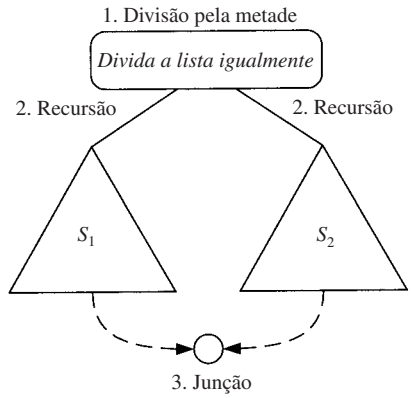


Figura 4.1 Um esquema visual do algoritmo *merge-sort*.

Mostramos um esquema do algoritmo *merge-sort* na Figura 4.1. Podemos visualizar a execução do algoritmo *merge-sort* usando uma árvore binária T, chamada de ***árvore merge-sort*** (veja a Figura 4.2.). Cada nodo de T representa uma invocação recursiva (ou chamada) do algoritmo *merge-sort*. Associamos com cada nodo v de T a seqüência S que é processada pela invocação associada com v. Os filhos do nodo v são associados com as chamadas recursivas que processam as subseqüências S_1 e S_2 de S. Os nodos externos de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.

A Figura 4.2 resume uma execução do algoritmo *merge-sort*, mostrando as seqüências de entrada e saída processadas em cada nodo da árvore *merge-sort*. Essa visualização do algoritmo em termos da árvore *merge-sort* ajuda a analisar o tempo de execução do algoritmo *merge-sort*. Em especial, uma vez que o tamanho da seqüência de entrada é grosseiramente dividido pela metade, a cada chamada recursiva do *merge-sort*, a altura da árvore *merge-sort* se aproxima de log n (lembramos que a base de log é 2, se omitida).

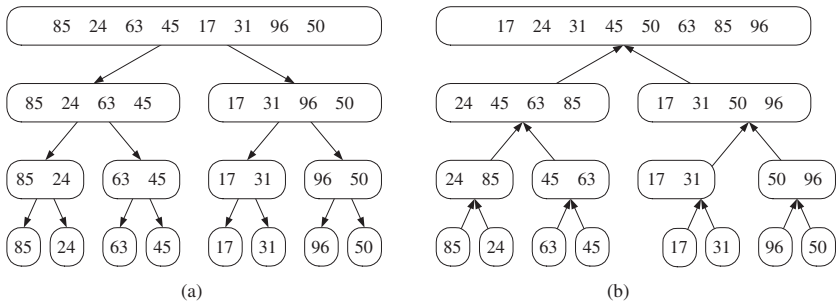


Figura 4.2 Árvore *merge-sort* T correspondente a uma execução do algoritmo *merge-sort* a partir de uma seqüência de oito elementos: (a) seqüências de entrada processadas em cada nodo de T; (b) seqüências de saída geradas em cada nodo de T.

No que se refere à etapa de divisão, lembramos que a notação $\lceil x \rceil$ indica o ***teto*** de x, ou seja, o menor inteiro m que satisfaz $x \leq m$. Da mesma forma, a notação $\lfloor x \rfloor$ indica o ***piso*** de x, ou seja, o maior inteiro k que satisfaz $k \leq x$. Assim, a etapa de divisão divide a lista S o mais igualmente possível, resultando no que segue.

Teorema 4.1: *a árvore merge-sort associada com uma execução do merge-sort a partir de uma seqüência de tamanho n tem altura* $\lceil \log n \rceil$.

Deixamos a justificativa do Teorema 4.1 como um simples exercício (R-4.1).

A partir de uma visão geral do *merge-sort* e da ilustração de seu funcionamento, vamos considerar cada um dos passos deste algoritmo de divisão e conquista em mais detalhes. Os passos de divisão e recursão do algoritmo *merge-sort* são simples: dividir uma seqüência de tamanho n envolve separá-la no elemento de ordem $\lceil n/2 \rceil$ e as chamadas recursivas compreendem simplesmente passar essas seqüências menores como parâmetro. O passo difícil é o de conquista, que faz a junção de duas seqüências ordenadas em uma única.

Junção de duas seqüências ordenadas

O algoritmo merge, no Algoritmo 4.3, faz a junção de duas seqüências ordenadas, S_1 e S_2, removendo iterativamente o menor elemento das duas e acrescentando-o no final da seqüência resultante, S, até que uma das duas seqüências esteja vazia, momento a partir do qual copiamos o restante da outra seqüência para a resultante.

Algoritmo merge(S_1, S_2, S):
  ***Entrada*:** seqüências S_1 e S_2 ordenadas de forma não-decrescente e uma seqüência S vazia.
  ***Saída*:** seqüência S contendo os elementos de S_1 e S_2 ordenadas de forma não-decrescente com as seqüências S_1 e S_2 tornando-se vazias
  enquanto (**não**(S_1. IsEmpty(). **ou** S_2.isEmpty) **faça**
    se S_1.first().element() $\leq$ S_2.first().element() **então**
      {remova o primeiro elemento de S_1 para o final de S}
      S.insertLast(S_1.remove(S_1.first()))
    senão
      {remova o primeiro elemento de S_2 para o final de S}
      S.insertLast(S_2.remove(S_2.first()))
  {remova os elementos restantes de S_1 para S}
  enquanto (**não** S_1.isEmpty()) **faça**
    S.insertLast(S_1.remove(S_1.first()))
  {remova os elementos restantes de S_2 para S}
  enquanto (**não** S_2.isEmpty()) **faça**
    S.insertLast(S_2.remove(S_2.first()))

Algoritmo 4.3 Algoritmo merge para junção de duas seqüências ordenadas.

Analisamos o tempo de execução do algoritmo merge fazendo algumas simples observações. Sejam n_1 e n_2 o número de elementos de S_1 e S_2, respectivamente. Vamos admitir, também, que as seqüências S_1 e S_2 e S são implementadas de forma que acesso, inserção e remoção de suas posições inicial e final consumam tempo $O(1)$. Este é o caso

para implementações baseadas em arranjos circulares ou listas duplamente encadeadas (Seção 2.2.3). O algoritmo merge tem três laços de **enquanto**. Devido às premissas, as operações executadas dentro de cada laço, consomem tempo $O(1)$. O ponto chave é que, durante cada iteração de um dos laços, um elemento é removido tanto de S_1 como de S_2. Uma vez que não são executadas inserções em S_1 ou S_2, esta observação implica que o número total de iterações dos três laços é $n_1 + n_2$. Em conseqüência, o tempo de execução do algoritmo merge é $O(n_1 + n_2)$, como podemos resumir:

Teorema 4.2: *a junção de duas seqüências ordenadas S_1 e S_2 consome tempo $O(n_1 + n_2)$, onde n_1 é o tamanho de S_1, e n_2 é o tamanho de S_2.*

Apresentamos um exemplo de execução do algoritmo merge na Figura 4.4.

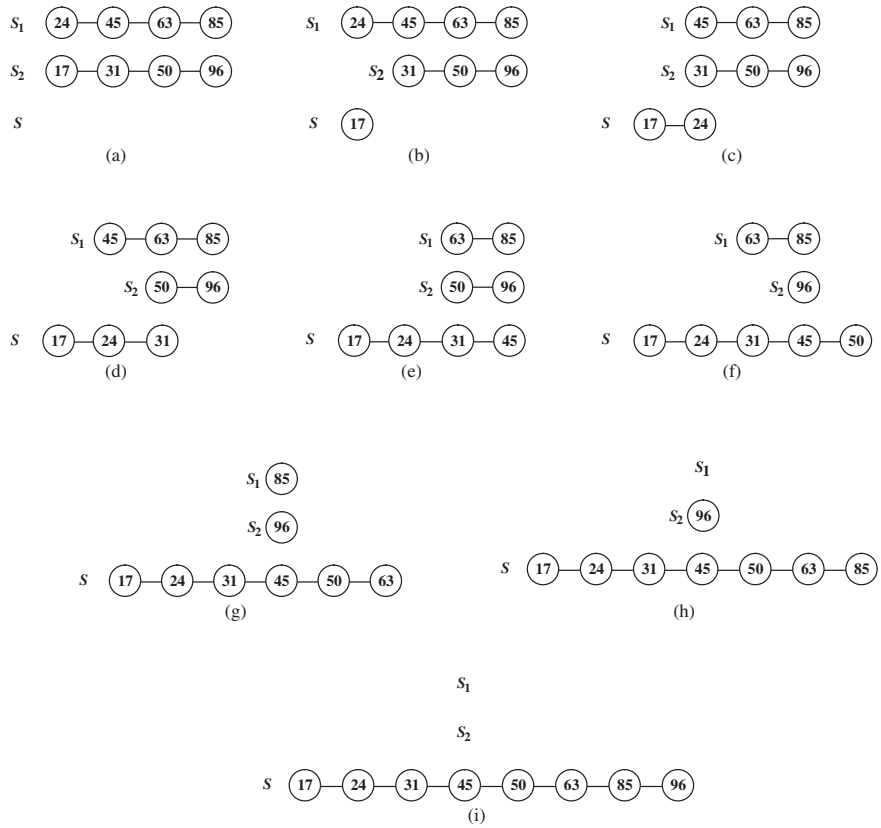


Figura 4.4 Exemplo de execução do algoritmo merge apresentado no Algoritmo 4.3.

O tempo de execução do *merge-sort*

Agora que temos os detalhes do algoritmo *merge-sort*, analisemos o tempo de execução do algoritmo *merge-sort* completo, considerando que é fornecida uma seqüência

de entrada com n elementos. Para facilitar, assumimos que n é potência de 2. No Exercício R-4.4 mostraremos que o resultado de nossa análise também pode ser aplicado quando n não for potência de 2.

Analisamos a rotina *merge-sort* referenciando a árvore *merge-sort* T. Chamamos o ***tempo gasto em um nodo*** v de T o tempo de execução de uma chamada recursiva associada com v, excluindo o tempo gasto esperando pelas chamadas recursivas associadas com os filhos de v para terminar. Em outras palavras, o tempo gasto no nodo v inclui os tempos de execução dos passos de divisão e conquista, mas exclui o tempo de execução do passo de recursão. Já observamos que os detalhes do passo de divisão são diretos; esse passo executa em tempo proporcional ao tamanho da seqüência para v. Da mesma forma, como mostrado no Teorema 4.2, o passo de conquista, que consiste na junção de duas subseqüências, consome tempo linear. Ou seja, fazendo i denotar a profundidade de um nodo v, o tempo gasto no nodo v é $O(n/2^i)$, uma vez que o tamanho da seqüência manipulada pela chamada recursiva associada com v é igual a $n/2^i$.

Examinando a árvore T de forma mais global, como mostrado na Figura 4.5, vemos que, dada nossa definição de "tempo gasto em um nodo", o tempo de execução do *merge-sort* é igual à soma dos tempos gastos nos nodos de T. Notamos que T tem exatamente 2^i nodos na profundidade i. Essa observação simples tem uma conseqüência importante, pois implica que o tempo total gasto em todos os nodos de T na profundidade i é $O(2^i \cdot n/2^i)$, o que corresponde a $O(n)$. Pelo Teorema 4.1, a altura de T é $\lceil \log n \rceil$. Em conseqüência, uma vez que o tempo gasto em cada um dos $\log n + 1$ níveis de T é $O(n)$, teremos o seguinte resultado:

Teorema 4.3: *merge-sort executa em* $O(n \log n)$, *no pior caso.*

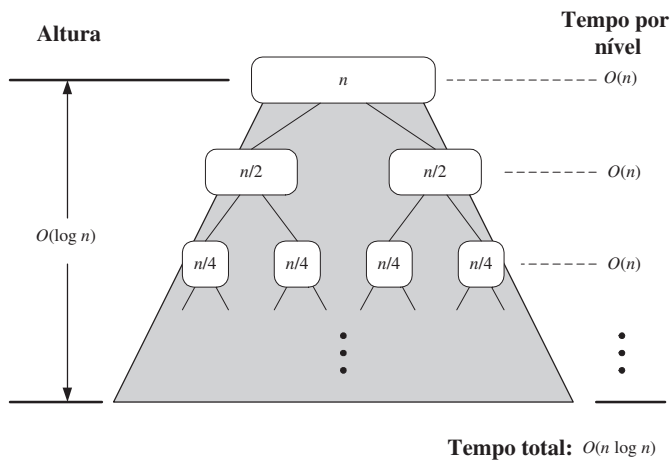


Figura 4.5 Análise visual do tempo de execução de *merge-sort*. Cada nodo da árvore *merge-sort* é rotulado com tamanho do subproblema correspondente.

4.1.2 O *merge-sort* e suas ações de recorrência

Existe outra forma de justificar que o tempo de execução do algoritmo *merge-sort* é $O(n \log n)$. Façamos a função $t(n)$ denotar o pior caso no que diz respeito ao tempo de execução do *merge-sort* em uma seqüência de entrada de tamanho n. Uma vez que o *merge-sort* é recursivo, podemos caracterizar a função $t(n)$ em termos das seguintes igualdades, onde a função $t(n)$ é expressa recursivamente em termos de si mesma, como segue:

$$t(n) = \begin{cases} b & \text{se } n = 1 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{nos demais casos} \end{cases}$$

onde $b > 0$ e $c > 0$ são constantes. Uma caracterização da função como a apresentada anteriormente é chamada de ***relação de recorrência*** (Seções 1.1.4 e 5.2.1), uma vez que a função aparece tanto do lado esquerdo como direito da igualdade. Apesar desta caracterização ser correta e precisa, o que realmente queremos é uma caracterização mais evidente do tipo O de $t(n)$ que não envolva a própria função $t(n)$ (ou seja, desejamos uma caracterização de forma fechada para $t(n)$).

De maneira a prover uma caracterização de forma fechada para $t(n)$, vamos restringir nossa atenção ao caso onde n é potência de 2. Deixamos o problema de mostrar que nossa caracterização assintótica ainda é válida, no caso geral, como exercício (R-4.4). Nesse caso, podemos simplificar a definição de $t(n)$ para

$$t(n) = \begin{cases} b & \text{se } n = 1 \\ 2t(n/2) + cn & \text{nos demais casos} \end{cases}$$

Mas, mesmo assim, ainda devemos tentar caracterizar essa equação recorrente de forma fechada. Uma maneira de fazer isso é aplicar iterativamente esta equação, assumindo que n é relativamente grande. Por exemplo, após mais uma aplicação desta equação podemos escrever uma nova recorrência para $t(n)$, como segue:

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2 t(n/2^2) + 2cn. \end{aligned}$$

Se aplicarmos a equação novamente, obtemos

$$t(n) = 2^3 t(n/2^3) + 3cn.$$

Após aplicar essa equação i vezes, obtemos

$$t(n) = 2^i t(n/2^i) + icn.$$

O ponto que resta, então, é determinar quando parar este processo. Para isso, lembramos que trocamos para a forma fechada $t(n) = b$ quando $n = 1$, o que irá ocorrer quando $2^i = n$. Em outras palavras, irá ocorrer quando $i = \log n$. Procedendo essa substituição, temos

$$\begin{aligned} t(n) &= 2^{\log n} t(n / 2^{\log n}) + (\log n)cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n. \end{aligned}$$

Isto é, obtemos uma justificativa alternativa para o fato de que $t(n)$ é $O(n \log n)$.

4.2 O TAD conjunto

Nesta seção, introduzimos o TAD ***conjunto***. Um ***conjunto*** é um contêiner que armazena objetos distintos. Ou seja, não existem elementos em duplicata em um conjunto, e não existe noção explícita de chave ou mesmo de ordem. Mesmo assim, incluímos a discussão sobre conjuntos neste capítulo sobre ordenação, porque a idéia de ordenação pode ter um papel importante em implementações eficientes das operações de um TAD conjunto.

Conjuntos e alguns de seus usos

Primeiramente, lembramos as definições matemáticas de ***união***, ***interseção*** e ***subtração*** de dois conjuntos A e B:

$$\begin{aligned} A \cup B &= \{x : x \in A \text{ ou } x \in B\} \\ A \cap B &= \{x : x \in A \text{ e } x \in B\} \\ A - B &= \{x : x \in A \text{ e } x \notin B\} \end{aligned}$$

Exemplo 4.4: *a maioria das máquinas de busca para Internet armazena, para cada palavra x do seu banco de dados de palavras, um conjunto W(x) de páginas WWW que contém x, onde cada página WWW é identificada por um único endereço Internet. Quando consultada sobre uma palavra x, tal máquina de busca necessita apenas retornar as páginas WWW presentes no conjunto W(x), ordenadas de acordo com alguma prioridade relativa à "importância" da página. Quando, por outro lado, é consultada sobre duas palavras x e y, a máquina de busca deve primeiramente calcular a interseção W(x) ∩ W(y) e, então, retornar as páginas presentes no conjunto resultante, ordenadas por prioridade. Muitas máquinas de busca usam o algoritmo descrito nesta seção para executar essa interseção.*

Métodos fundamentais do TAD conjunto

Os métodos fundamentais do TAD conjunto, que atuam sobre um conjunto A, são os seguintes:

union(B): substitui A pela união de A e B, ou seja, executa $A \leftarrow A \cup B$.

intersect(B): substitui A pela interseção de A e B, ou seja, executa $A \leftarrow A \cap B$.

subtract(B): substitui A pela diferença entre A e B, ou seja, executa $A \leftarrow A - B$.

Definimos as operações union, intersect e subtract acima de maneira que alteram o conteúdo do conjunto A envolvido. Em vez disso, podemos modificar essas operações para que não modifiquem A, e sim retornem um conjunto novo.

4.2.1 Uma implementação simples para conjunto

Uma das formas mais simples de se implementar um conjunto é armazenar seus elementos em uma seqüência ordenada. Essa implementação é incluída, por exemplo, em várias bibliotecas de *software* para estruturas de dados genéricas. Sendo assim, vamos considerar a implementação do TAD conjunto usando uma seqüência ordenada (consideraremos outras implementações em vários exercícios). Qualquer relação de ordem total consistente entre os elementos do conjunto pode ser usada, desde que a mesma ordem seja usada para todos os conjuntos.

Implementaremos cada uma das três operações fundamentais de um conjunto usando uma versão genérica do algoritmo de junção que recebe, como entrada, duas seqüências ordenadas representando os conjuntos de entrada e gera uma seqüência indicando o conjunto de saída, seja ele a união, interseção ou diferença dos conjuntos de entrada.

O método genérico de junção examina e compara iterativamente os elementos correntes a e b das seqüências A e B, respectivamente, e determina quando $a < b$, $a = b$ ou $a > b$. Então, baseado no resultado desta comparação, determina se pode copiar um ou nenhum dos elementos a e b para o fim da seqüência de saída C. Essa determinação é feita baseada na operação específica que estamos executando, seja união, interseção ou diferença. O próximo elemento de uma ou ambas as seqüências é então considerado. Por exemplo, em uma operação de união, procedemos como segue:

- se $a < b$, copiamos a para a seqüência de saída e avançamos para o próximo elemento de A;
- se $a = b$, copiamos a para a seqüência de saída e avançamos para o próximo elemento de A e B;
- se $a > b$, copiamos b para a seqüência de saída e avançamos para o próximo elemento de B.

Desempenho da junção genérica

Analisamos o tempo de execução do algoritmo genérico de junção. A cada iteração, comparamos dois elementos das seqüências de entrada A e B, possivelmente copiando um elemento para a seqüência de saída, e avançamos o elemento corrente de A, B ou ambos. Admitindo que as comparações e cópias levam tempo $O(1)$, o tempo total de execução é $O(n_A + n_B)$, onde n_A é o tamanho de A e n_B é o tamanho de B; ou seja, a junção leva um tempo proporcional ao número de elementos envolvidos. Desta forma temos:

Teorema 4.5: *O TAD conjunto pode ser implementado usando um esquema de seqüência ordenada e junção genérica que suporta as operações* union, intersect e subtract *em tempo O(n), onde n indica a soma dos tamanhos dos conjuntos envolvidos.*

Existe também uma versão especial e importante do TAD conjunto que se aplica somente quando lidamos com coleções provenientes de conjuntos disjuntos.

4.2.2 Partições com operações *union* e *find*

Uma ***partição*** é uma coleção de conjuntos disjuntos. Definimos os métodos de um TAD partição usando localizadores (Seção 2.4.4) para acessar os elementos armazenados em um conjunto. Cada localizador, neste caso, funciona como um ponteiro, provendo acesso imediato à posição (nodo) onde o elemento está armazenado na partição.

makeSet(e): cria um conjunto único contendo o elemento e e retorna um localizador ℓ para e.

union(A, B): calcula e retorna conjunto $A \leftarrow A \cup B$.

find(ℓ): retorna o conjunto contendo o elemento com localizador ℓ.

Implementação usando seqüência

Uma implementação simples de uma partição com um total de n elementos dá-se com uma coleção de seqüências, uma para cada conjunto, onde a seqüência para o conjunto A armazena nodos localizadores como seus elementos. Cada nodo localizador tem uma referência para seu elemento e, o qual permite a execução do método element() do TAD localizador em tempo $O(1)$ e uma referência para a seqüência que armazena e (veja a Figura 4.6). Assim, podemos executar a operação find(ℓ) em tempo $O(1)$. Da mesma forma, makeSet também consome tempo $O(1)$. A operação union(A,B) requer que unamos as duas seqüências em uma e atualizemos a seqüência de referências dos localizadores de uma das duas. Escolhemos implementar essa operação removendo todos os localizadores da seqüência de menor tamanho e inserindo-os na seqüência de maior tamanho. Desta forma, a operação union(A,B) consome tempo $O(\min(|A|,|B|))$, o que é $O(n)$ porque, no pior caso, $|A| = |B| = n/2$. Entretanto, como pode ser visto a seguir, uma análise amortizada nos mostra que essa implementação é muito melhor do que aparenta a partir da análise do pior caso.

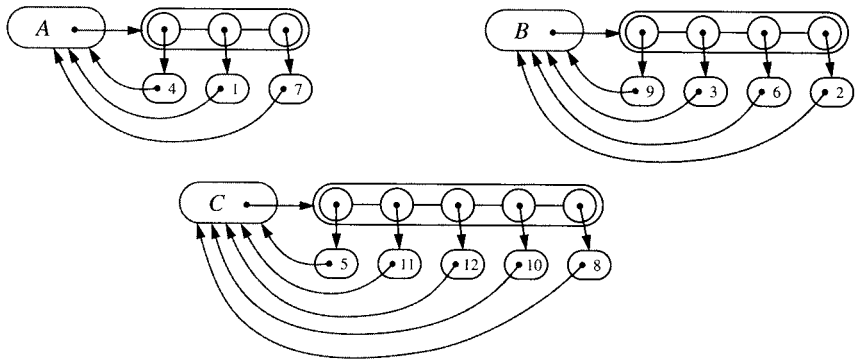


Figura 4.6 Implementação baseada em seqüência de uma partição composta de três conjuntos: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$ e $C = \{5, 8, 10, 11, 12\}$.

Desempenho da implementação usando seqüência

A implementação usando seqüência é simples e também eficiente, como demonstrado pelo seguinte teorema:

Teorema 4.6: *a execução de uma série de n operações* makeSet, union *e* find *usando a implementação baseada em seqüência, a partir de uma partição inicialmente vazia, consome tempo* $O(n \log n)$.

Prova: usamos o método da conta corrente assumindo que um ciberdólar paga pelo tempo para executar uma operação find, uma operação makeSet ou o movimento de um nodo localizador de uma seqüência para outra em uma operação de union. No caso de uma operação find ou makeSet, cobramos pela operação em si um ciberdólar. No caso de uma operação union, entretanto, cobramos um ciberdólar por cada nodo localizador que movemos de um conjunto para o outro. Não cobramos nada pelas operações union propriamente ditas. Naturalmente, o total cobrado pelas operações find e makeSet é $O(n)$.

Consideremos, então, o número de cobranças feitas para os localizadores em função de operações union. A observação importante é que, cada vez que movemos um localizador de um conjunto para outro, o tamanho do novo conjunto, no mínimo, dobra. Assim, cada localizador é movido de um conjunto para outro no máximo $\log n$ vezes; logo, cada localizador pode ser cobrado no máximo $O(\log n)$ vezes. Uma vez que assumimos que a partição está inicialmente vazia, existem $O(n)$ elementos diferentes referenciados na série de operações fornecida, o que implica que o tempo total para todas as operações union é $O(n \log n)$. ■

O tempo amortizado de execução de uma operação em uma série de operações makeSet, union e find corresponde ao tempo total consumido pela série dividido pelo número de operações. Concluímos do teorema anterior que, para uma partição implementada usando seqüências, o tempo amortizado e execução de cada operação é $O(\log n)$. Assim, podemos resumir o desempenho de nossa implementação simples de partição baseada em seqüência como segue.

Teorema 4.7: *usando uma implementação baseada em seqüência de uma partição, em uma série de n operações* makeSet, union *e* find *iniciando em uma partição inicialmente vazia, o tempo amortizado de execução de cada operação é* $O(\log n)$.

Observamos que, nesta implementação baseada em seqüência de uma partição, cada operação find, de fato, consome tempo $O(1)$, no pior caso. Esse é o tempo de execução das operações union, que são o gargalo computacional.

Na próxima seção, descreveremos uma implementação de partição baseada em árvore que não garante tempo constante para as operações find, mas tem tempo amortizado muito melhor do que $O(\log n)$ para a operação union.

4.2.3 Uma implementação de partição usando árvores

Uma estrutura alternativa para implementar uma partição com n elementos usa uma coleção de árvores para armazenar os elementos nos conjuntos, onde cada árvore está associada com um conjunto diferente (veja a Figura 4.7). Implementamos uma árvore T com uma estrutura de dados encadeada, onde cada nodo u de T armazena um elemento do conjunto associado com T e uma referência pai apontando para o nodo pai de u. Se u é a raiz, então a referência pai aponta para si mesma. Os nodos da árvore ser-

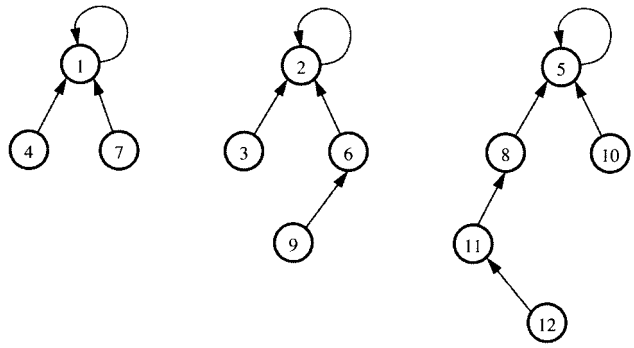


Figura 4.7 Implementação baseada em árvore de uma partição composta de três conjuntos distintos: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$ e $C = \{5, 8, 10, 11, 12\}$.

vem como localizadores dos elementos na partição. Além disso, identificamos cada conjunto com a raiz da árvore associada.

Usando essa estrutura de dados para a partição, a operação union é executada transformando uma das duas árvores em subárvore da outra (Figura 4.8b), o que pode ser feito em tempo $O(1)$ atribuindo a referência pai da raiz de uma árvore para apontar para a raiz da outra árvore. A operação find para o nodo localizador ℓ é executada caminhando pela árvore em direção à raiz da árvore que contém o localizador ℓ (Figura 4.8a), o que consome tempo $O(n)$, no pior caso.

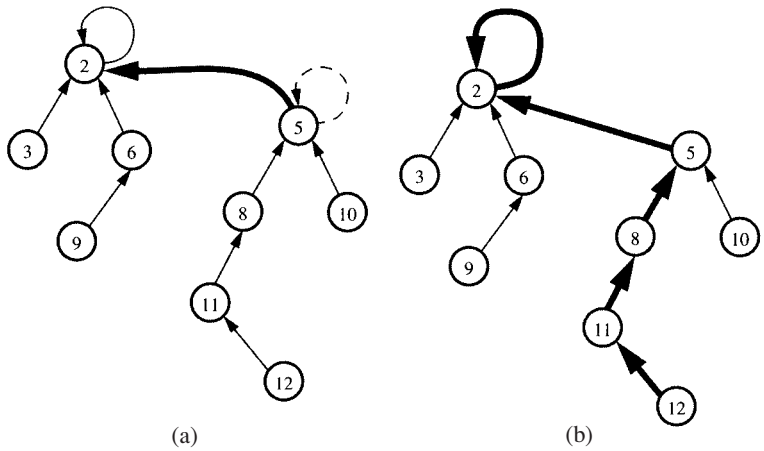


Figura 4.8 Implementação baseada em árvore de uma partição: (a) operação union(A,B); (b) operação find(ℓ), onde ℓ denota o localizador do elemento 12.

Observamos que essa representação de uma árvore é uma estrutura de dados especializada usada para implementar uma partição, e isso não significa que seja uma implementação do tipo abstrato de dados árvore (Seção 2.3). Na verdade, essa representação tem apenas conexões "para cima" e não provê uma maneira de acessar os filhos de um determinado nodo. Em um primeiro momento, essa implementação parece não ser melhor do que uma estrutura de dados baseada em uma seqüência, mas acrescentamos as seguintes heurísticas simples que a fazem ser mais rápida:

União pelo tamanho: armazena-se em cada nodo v o tamanho da subárvore enraizada em v, denotado por $n(v)$. Em uma operação union, a árvore do menor conjunto é transformada em subárvore da outra, e atualiza-se o campo tamanho da raiz da árvore resultante.

Compressão de caminhos: em uma operação find, para cada nodo v visitado pelo find, inicializa-se o ponteiro pai de v apontando o mesmo para a raiz (veja a Figura 4.9).

Essas heurísticas aumentam o tempo de execução de uma operação através de um fator constante, mas, como mostramos a seguir, melhoram significativamente o tempo amortizado de execução.

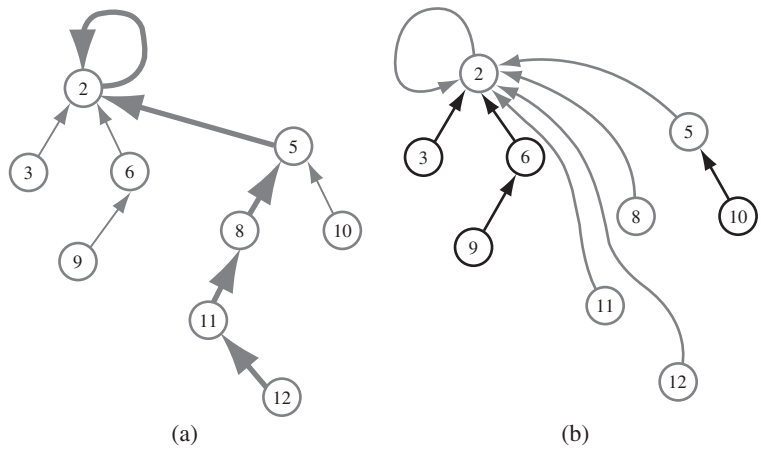


Figura 4.9 Heurística de compressão de caminhos: (a) caminho percorrido pela operação find sobre o elemento 12; (b) árvore reestruturada.

Definição de uma função de colocação

Vamos analisar o tempo de execução de uma série de n operações find e union em uma partição constituída inicialmente de n conjuntos com apenas um elemento.

Para cada nodo v que é raiz, lembramos que definimos $n(v)$ como sendo o tamanho da subárvore enraizada em v (incluindo v) e que identificamos um conjunto contendo a raiz e sua árvore associada.

Atualizamos o campo tamanho de v cada vez que um conjunto é unido a v. Assim, se v não é uma raiz, então $n(v)$ corresponde à maior subárvore enraizada em v, o que ocorre imediatamente antes de aplicarmos a operação union de v com algum outro nodo cujo tamanho seja, no mínimo, tão grande quanto o de v. Para qualquer nodo v, definimos a ***colocação*** de v, denotada por $r(v)$, como

$$r(v) = \lfloor \log\ n(v) \rfloor.$$

Assim, obtemos de imediato $n(v) \geq 2r^{(v)}$. Também, já que existem no máximo n nodos na árvore de v, $r(v) \leq \lfloor \log n \rfloor$, para cada nodo v.

Teorema 4.8: *se o nodo w for pai do nodo v, então*

$$r(v) < r(w).$$

Prova: fazemos v apontar para w apenas se o tamanho de w após a união for, no mínimo, tão grande quanto v. Fazemos $n(w)$ denotar o tamanho de w antes da união, e $n'(w)$ denotar o tamanho de w após a união. Assim, após a união, teremos

$$\begin{aligned} r(v) &= \lfloor \log\ n(v) \rfloor \\ &< \lfloor \log\ n(v) + 1 \rfloor \\ &= \lfloor \log\ 2n(v) \rfloor \\ &\leq \lfloor \log\ (n(v) + n(w)) \rfloor \\ &= \lfloor \log\ n'(w) \rfloor \\ &\leq r(w) \end{aligned}$$

■

Dito de outra forma, esse teorema define que as colocações crescem ***monotonicamente*** à medida que seguimos os ponteiros pai árvore acima. Isso também implica o seguinte:

Teorema 4.9: *existem no máximo $n/2^s$ nodos na colocação s, para* $0 \leq s \leq \lfloor \log n \rfloor$.

Prova: pelo teorema anterior, $r(v) < r(w)$, para qualquer nodo v com pai w, e as colocações crescem monotonicamente à medida que subimos por qualquer árvore seguindo os ponteiros pai. Assim, se $r(v) = r(w)$ para dois nodos v e w, então os nodos contados em $n(v)$ devem ser separados e distintos dos nodos contados em $n(w)$. Pela definição de colocação, se um nodo v está em uma colocação s, então $n(v) \geq 2^s$. Portanto, uma vez que existem no máximo n nodos no total, pode haver no máximo $n/2^s$ que estão na mesma colocação s. ■

Análise amortizada

Uma propriedade surpreendente da estrutura de dados partição baseada em árvore, quando implementada usando as heurísticas de união pelo tamanho e compressão de caminhos, é que a executação uma série de operações union e find consome tempo $O(n \log^* n)$, onde $\log^* n$ é a função "logaritmo de iteração", que é a inversa da função ***torre-de-dois*** $t(i)$:

$$t(i) = \begin{cases} 1 & \text{se } i = 0 \\ 2^{t(i-1)} & \text{se } i \geq 1. \end{cases}$$

Ou seja,

$$\log^* n = \min\{i : t(i) \geq n\}.$$

Intuitivamente, $\log^* n$ é o número de vezes que alguém pode, iterativamente, calcular o logaritmo (base 2) de um número antes de obter um número menor do que 2. A Tabela 4.10 mostra algumas amostras de valores dessa função.

mínimo n	$\log^* n$
1	0
2	1
$2^2 = 4$	2
$2^{2^2} = 16$	3
$2^{2^{2^2}} = 65.536$	4
$2^{2^{2^{2^2}}} = 2^{65.536}$	5

Tabela 4.10 Amostras de valores de $\log^* n$ e o menor valor de n necessário para obter este valor.

Como demonstrado na Tabela 4.10, para todos os propósitos práticos, $\log^* n \leq 5$. Essa é uma função de crescimento surpreendentemente lento (mas que, no entanto, está crescendo).

De maneira a justificar nossa surpreendente declaração de que o tempo necessário para executar as operações union e find é $O(n \log^* n)$, dividimos os nodos em ***grupos de colocações***. Os nodos v e u estarão no mesmo grupo de colocações g se

$$g = \log^*(r(v)) = \log^*(r(u)).$$

Uma vez que a maior colocação é $\lfloor \log n \rfloor$, o maior grupo de colocações é

$$\log^*(\log n) = \log^* n - 1.$$

Usamos grupos de colocações para derivar regras para uma análise amortizada através de um método de conta corrente. Já observamos que a execução de uma operação union consome tempo $O(1)$. Taxamos cada operação union com um ciberdólar para pagar por ela. Assim podemos nos concentrar nas operações find para justificar nossa afirmação de que executar todas as n operações consome tempo $O(n \log^* n)$.

A principal tarefa computacional na execução de uma operação find é seguir os ponteiros pai a partir de um nodo u em direção à raiz da árvore que contém u. Podemos contabilizar esse trabalho pagando um ciberdólar por cada referência pai que percorremos. Seja v um nodo ao longo desse caminho, e seja w o pai de v. Usamos duas regras para cobrar a travessia dessa referência pai:

- se w for a raiz ou se w estiver em um grupo de colocações diferente de v, então cobramos um ciberdólar pela operação find.
- em qualquer outro caso (w não é raiz, e v e w estão no mesmo grupo de colocações), cobramos do nodo v um ciberdólar.

Uma vez que existem no máximo $\log^* n - 1$ grupos, essa regra garante que qualquer operação find seja cobrada no máximo de $\log^* n$ ciberdólares. Assim, esse esquema faz a contabilização para a operação find, mas ainda precisamos contabilizar todos os ciberdólares que cobramos dos nodos.

Observamos que, após cobrarmos um nodo v, então v terá um novo pai, que é um nodo mais acima na árvore de v. Além disso, uma vez que as colocações crescem monotonicamente à medida que se sobe pela árvore, a colocação do novo pai de v será maior do que a colocação do antigo pai de v, ou seja, w. Assim, qualquer nodo v pode ser cobrado no máximo o número de diferentes colocações do seu grupo. Também, uma vez que qualquer nodo no grupo de colocações 0 tem pai em um grupo de colocações maior, podemos restringir nossa atenção aos nodos nos grupos maiores do que 0 (para os quais sempre cobraremos a operação find para examinar um nodo no grupo 0). Se v estiver em um grupo $g > 0$, então v pode ser cobrado no máximo $t(g) - t(g-1)$ vezes antes que v tenha um pai em um grupo de maior ordem (e, deste ponto em diante, v nunca será cobrado novamente). Em outras palavras, o número total C de ciberdólares que sempre são cobrados dos nodos pode ser limitado por

$$C \leq \sum_{g=1}^{\log^* n - 1} n(g) \cdot (t(g) - t(g-1)),$$

onde $n(g)$ indica a quantidade de nodos no grupo de colocações g.

Portanto, se derivarmos um limite superior para $n(g)$, poderemos conectá-lo na equação acima para derivar um limite superior para C. Para derivar o limite de $n(g)$, o número de nodos no grupo de colocações g, lembramos que o número total de nodos de uma dada colocação s é de no máximo $n/2^s$ (pelo Teorema 4.9). Assim, para $g > 0$,

$$\begin{aligned}
n(g) &\leq \sum_{s=t(g-1)+1}^{t(g)} \frac{n}{2^s} \\
&= \frac{n}{2^{t(g-1)+1}} \sum_{s=0}^{t(g)-t(g-1)-1} \frac{1}{2^s} \\
&< \frac{n}{2^{t(g-1)+1}} \cdot 2 \\
&= \frac{n}{2^{t(g-1)}} \\
&= \frac{n}{t(g)} \cdot
\end{aligned}$$

Conectando esse limite no limite fornecido anteriormente para o número total de cobranças de um nodo C, teremos

$$
\begin{aligned}
C &< \sum_{g=1}^{\log^* n - 1} \frac{n}{t(g)} \cdot (t(g) - t(g-1)) \\
&\leq \sum_{g=1}^{\log^* n - 1} \frac{n}{t(g)} \cdot t(g) \\
&= \sum_{g=1}^{\log^* n - 1} n \\
&\leq n \ \log^* n.
\end{aligned}
$$

Portanto, mostramos que o total de ciberdólares cobrado dos nodos é de no máximo $n \log^* n$. Isto implica o seguinte:

Teorema 4.10: *seja P uma partição com n elementos implementada usando uma coleção de árvores, como descrito antes, utilizando as heurísticas união por tamanho e compressão de caminhos. Em uma série de operações* union e find *executadas sobre P, começando com uma coleção contendo conjuntos de apenas um elemento, o tempo amortizado de execução de cada operação é* $O(\log^* n)$.

A função de Ackermann

Na verdade, podemos provar que o tempo amortizado de execução de uma operação em uma série de n operações de partição implementadas como anteriormente é $O(\alpha(n))$, onde $\alpha(n)$ é uma função chamada de inversa da ***função de Ackermann***, $\mathcal{A}$, que cresce assintoticamente mais lentamente do que $\log^* n$, mas provar isso está fora do escopo deste livro.

Por fim, definimos aqui a função de Ackermann para poder apreciar quão rápido ela cresce; logo, quão devagar cresce sua inversa. Primeiramente definimos uma função de Ackermann indexada $\mathcal{A}_i$, como segue:

$$
\begin{aligned}
&\mathcal{A}_0(n) = 2n && \text{para } n \geq 0 \\
&\mathcal{A}_i(1) = \mathcal{A}_{i-1}(2) && \text{para } i \geq 1 \\
&\mathcal{A}_i(n) = \mathcal{A}_{i-1}(A_i(n-1)) && \text{para } i \geq 1 \text{ e } n \geq 2
\end{aligned}
$$

Em outras palavras, a função indexada de Ackermann define uma progressão de funções com cada função crescendo mais rapidamente do que a anterior:

- $\mathcal{A}_0(n) = 2n$ é a função de multiplicação por 2;
- $\mathcal{A}_1(n) = 2^n$ é a função de potência de 2;
- $\mathcal{A}_2(n) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (com n 2) é a função torre-de-dois;
- $\mathcal{A}_3(n)$ é a função torre-de-torre-de-dois;
- e assim por diante.

Definimos então a função de Ackermann como $\mathcal{A}(n) = \mathcal{A}_n(n)$, que é uma função de crescimento incrivelmente rápido. Da mesma forma, sua inversa, $\alpha(n) = \min\{m: \mathcal{A}(m) \geq n\}$, é uma função de crescimento incrivelmente lento.

Em seguida, retornaremos ao problema de ordenação discutindo o ***quick-sort***. Da mesma forma que o *merge-sort*, este algoritmo também é baseado no paradigma de ***divisão e conquista***, mas usa esta técnica ao contrário, na medida em que todo trabalho pesado é feito ***antes*** das chamadas recursivas.

4.3 *Quick-sort*

O algoritmo *quick-sort* ordena uma seqüência S usando uma abordagem de divisão e conquista em que dividimos S em subseqüências, aplicando recursão para ordenar cada subseqüência e, então, combinar as subseqüências ordenadas por concatenação simples. Detalhando, o algoritmo *quick-sort* consiste nos três passos seguintes (veja a Figura 4.11):

1. ***Divisão:*** se S tiver pelo menos dois elementos (nada precisa ser feito se S tiver zero ou um elemento), escolha um elemento x de S, chamado de ***pivô***. Normalmente, escolhemos como pivô x o último elemento de S. Removemos todos os elementos de S e os colocamos em três seqüências:
   - L, armazenando os elementos de S menores do que x;
   - E, armazenando os elementos de S iguais a x;
   - G, armazenando os elementos de S maiores do que x.

   (Se os elementos de S são todos diferentes, então E armazena apenas um elemento – o próprio pivô.)
2. ***Recursão:*** ordene as seqüências L e G, recursivamente.
3. ***Conquista:*** coloque de volta os elementos em S, em ordem, inserindo primeiramente os elementos de L, em seguida os de E e, por fim, os de G.

Da mesma forma que o *merge-sort*, podemos visualizar a execução do *quick-sort* usando uma árvore binária recursiva chamada ***árvore quick-sort***. A Figura 4.12 visualiza a execução do algoritmo *quick-sort*, mostrando seqüências de exemplos de entrada e saída processadas em cada nodo da árvore *quick-sort*.

Ao contrário do *merge-sort*, entretanto, a altura da árvore *quick-sort* associada com a execução do *quick-sort* é linear no pior caso. Isso acontece, por exemplo, quando a seqüência consiste em n elementos distintos e já está ordenada. Na verdade, neste caso, a escolha padrão do maior elemento para pivô produz uma subseqüência L de tamanho $n - 1$, enquanto a subseqüência E tem tamanho 1, e a subseqüência G tem tamanho 0. Desta forma, a altura da árvore *quick-sort* é $n - 1$ no pior caso.

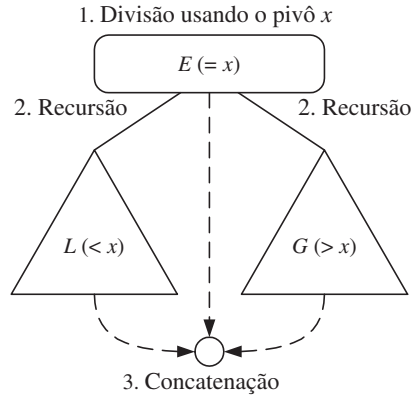


Figura 4.11 Visualização esquemática do algoritmo *quick-sort*.

Tempo de execução do *quick-sort*

Podemos analisar o tempo de execução do *quick-sort* com a mesma técnica usada para o *merge-sort* na Seção 4.1.1. Na verdade, identificamos o tempo gasto em cada nodo da árvore de *quick-sort* T (Figura 4.12) e somamos os tempos de execução para todos os nodos. Os passos de divisão e conquista do *quick-sort* são fáceis de implementar em tempo linear. Desta forma, o tempo gasto no nodo v de T é proporcional ao ***tamanho da entrada*** $s(v)$ de v, definido como o tamanho da seqüência manipulada pela invocação do *quick-sort* associada com o nodo v. Uma vez que a subseqüência E tem pelo menos um elemento (o pivô), a soma dos tamanhos das entradas dos filhos de v é no máximo $s(v) - 1$.

Dada uma árvore *quick-sort* T, faça s_i indicar a soma dos tamanhos de entrada dos nodos na profundidade i de T. Claramente, $s_0 = n$, uma vez que a raiz r de T está associada com a seqüência completa. Da mesma forma, $s_1 \leq n - 1$, uma vez que o pivô não é propagado para o filho de r. Consideremos em seguida s_2. Se os dois filhos de r tiverem tamanho de entrada diferente de zero, então $s_2 = n - 3$. Em qualquer outro caso (um filho da raiz tem tamanho zero, o outro tem tamanho $n - 1$), $s_2 = n - 2$. Desta forma, $s_2 \leq n - 2$. Continuando essa linha de raciocínio, obtemos $s_i \leq n - i$.

Como observado na Seção 4.3, a altura de T é $n - 1$, no pior caso. Sendo assim, o pior caso para o tempo de execução do *quick-sort* é

$$O\left(\sum_{i=0}^{n-1} s_i\right), \text{ que é } O\left(\sum_{i=0}^{n-1}(n-1)\right) \text{ isto é, } O\left(\sum_{i=1}^{n} i\right).$$

Pelo Teorema 1.13, $\sum_{i=1}^{n} i$ é $O(n^2)$. Logo, o *quick-sort* executa em tempo $O(n^2)$, no pior caso.

Considerando o nome, podemos esperar que o *quick-sort* execute depressa. Entretanto, os limites quadráticos mostrados indicam que o *quick-sort* é lento no pior caso. Paradoxalmente, esse comportamento do pior caso ocorre em situações problemáticas em que a ordenação é simples – se a seqüência já está ordenada. Ainda, observamos que o melhor caso do *quick-sort* sobre uma seqüência de elementos distintos ocorre

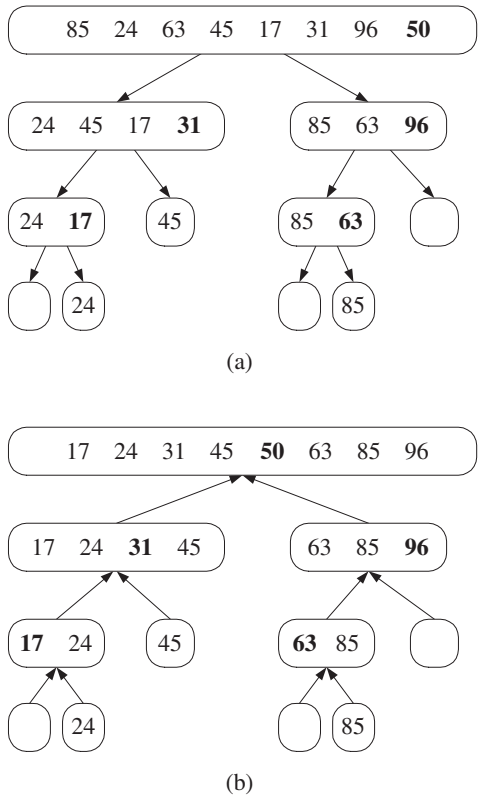


Figura 4.12 Árvore *quick-sort T* correspondente a uma execução do algoritmo *quick-sort* em uma seqüência de oito elementos: (a) seqüências de entrada processadas em cada nodo de *T*; (b) seqüências de saída geradas em cada nodo de *T*. O pivô usado em cada nível de recursão é mostrado em negrito.

quando as subseqüências L e G têm, aproximadamente, o mesmo tamanho. Na verdade, neste caso, preservamos um pivô em cada nodo interno e fazemos duas chamadas de mesmo tamanho para seus filhos. Assim, salvamos um pivô na raiz, dois no nível 1, 2^2 no nível 2, e assim por diante. Ou seja, no melhor caso temos

$$
\begin{aligned}
s_0 &= n \\
s_1 &= n - 1 \\
s_2 &= n - (1 + 2) = n - 3 \\
&\vdots \\
s_i &= n - (1 + 2 + 2^2 + \cdots + 2^i - 1) = n - (2^i - 1),
\end{aligned}
$$

e assim por diante. Logo, no melhor caso, T tem altura $O(\log n)$ e o *quick-sort* executa em tempo $O(n \log n)$; deixamos a justificativa deste fato para o Exercício R-4.11.

A intuição informal por trás do comportamento esperado do *quick-sort* é que, a cada invocação, provavelmente, o pivô irá dividir a seqüência de entrada em partes mais

ou menos iguais. Desta forma, esperamos que o tempo médio de execução do *quick-sort* seja semelhante ao tempo do melhor caso, ou seja, $O(n \log n)$. Na próxima seção, veremos que a introdução de randomização torna o comportamento do *quick-sort* exatamente o que acabamos de descrever.

4.3.1 *Quick-sort* randomizado

Uma forma normal de se analisar o *quick-sort* é supor que o pivô irá sempre dividir a seqüência em partes quase iguais. Essa premissa, entretanto, pressupõe um conhecimento sobre a distribuição da entrada que normalmente não está disponível. Por exemplo, teremos de assumir que raramente serão fornecidas seqüências "quase" ordenadas para pôr em ordem, o que pode ser normal em muitas aplicações. Por sorte, esta premissa não é necessária para casarmos nossa intuição com o comportamento do *quick-sort*.

Uma vez que o objetivo do passo de divisão do método *quick-sort* é dividir a seqüência S em partes quase iguais, vamos introduzir uma nova regra para tomar o pivô – escolher um ***elemento randômico*** da seqüência de entrada. Como mostraremos a seguir, o algoritmo resultante, chamado ***quick-sort randomizado***, tem um tempo esperado de execução de $O(n \log n)$, dada uma seqüência de n elementos.

Teorema 4.11: *o tempo esperado de execução para o quick-sort randomizado aplicado em uma seqüência de tamanho n é* $O(n \log n)$.

Prova: faremos uso de um fato simples originário da teoria da probabilidade:

> *O número esperado de vezes que uma moeda deve ser jogada até mostrar a face "cara" k vezes é* $2k$.

Consideremos agora uma invocação recursiva simples de um *quick-sort* randomizado e façamos m indicar o tamanho da seqüência de entrada para essa invocação. Digamos que essa invocação é "boa" se o pivô escolhido for tal que as subseqüências L e G tenham tamanho no mínimo $m/4$ e no máximo $3m/4$ cada. Desta forma, uma vez que o pivô é escolhido de forma uniformemente randômica e que existem $m/2$ pivôs para os quais a invocação é boa, a probabilidade de que a invocação seja boa é 1/2 (a mesma da moeda mostrar a face "cara").

Se um nodo v da árvore de *quick-sort* T, como mostrado na Figura 4.13, for associado com uma "boa" chamada recursiva, então os tamanhos das entradas dos filhos de v serão no máximo $3s(v)/4$ cada (que é o mesmo que $s(v)/(4/3)$). Se tomarmos qualquer caminho em T a partir da raiz em direção a um nodo externo, então o comprimento deste caminho será, no máximo, o número de invocações que têm de ser feitas (em cada nodo deste caminho) até obtermos $\log_{4/3} n$ boas invocações. Aplicando o fato probabilístico visto anteriormente, o número esperado de invocações que precisamos fazer até isto ocorrer é $2 \log_{4/3} n$ (se um caminho terminar antes deste nível, será melhor). Sendo assim, o comprimento esperado de qualquer caminho partindo da raiz até um nodo externo em T é $O(\log n)$. Lembrando que o tempo gasto em cada nível de T é $O(n)$, o tempo esperado de execução para o *quick-sort* randomizado é $O(n \log n)$. ■

Observamos que a expectativa para o tempo de execução é calculada a partir de todas as possíveis escolhas que o algoritmo faz e é independente de qualquer premissa sobre a distribuição das seqüências de entrada que o algoritmo pode encontrar. Na verdade, usando probabilidade, podemos mostrar que o tempo de execução do *quick-sort* randomizado é $O(n \log n)$, com alta possibilidade (veja o Exercício C-4.8.).

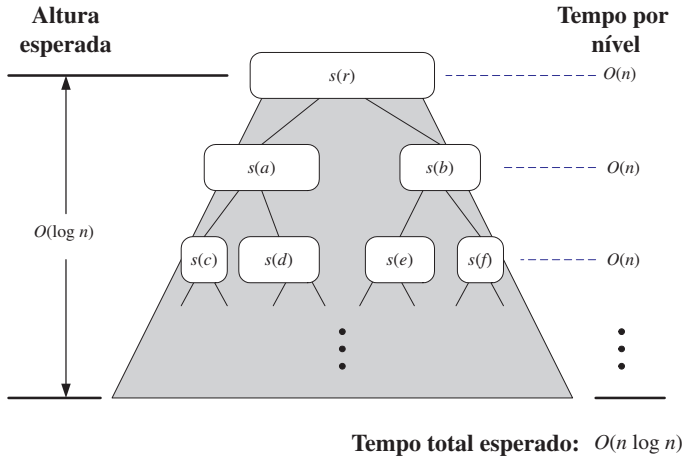


Figura 4.13 Análise visual do tempo da árvore *quick-sort T*.

4.4. Limite inferior em ordenação baseada em comparação

Retomando a discussão sobre ordenação, descrevemos vários métodos, tanto com o pior caso, como com o tempo esperado de execução $O(n \log n)$ para seqüências de entrada de tamanho n. Esses métodos incluem o *merge-sort* e o *quick-sort*, descritos neste capítulo, bem como o *heapsort*, descrito na Seção 2.4.4. A questão que surge naturalmente, então, é se é possível ordenar mais rapidamente que em tempo $O(n \log n)$.

Nesta seção, mostraremos que, se a operação primitiva computacional usada pelo algoritmo de ordenação é uma comparação de dois elementos, então isso é o melhor que podemos fazer – ordenação baseada em comparação tem o limite mínimo, no pior caso, da ordem de $\Omega(n \log n)$ relativamente ao tempo de execução. (Lembre-se da notação $\Omega(.)$ vista na Seção 1.2.2.) Concentrando-se no custo principal da ordenação baseada em comparação, vamos contar apenas as comparações que o algoritmo de ordenação executa. Como queremos derivar um limite mínimo, isso será suficiente.

Suponhamos que temos uma seqüência $S = (x_0, x_1,..., x_{n-1})$ que desejamos ordenar, e consideremos que todos os elementos de S são distintos (isso não é uma restrição, uma vez que estamos definindo um limite mínimo). Cada vez que o algoritmo de ordenação compara dois elementos x_i e x_j (ou seja, pergunta "se $x_i < x_j$?"), existem duas saídas possíveis: "sim" ou "não". Baseado no resultado desta comparação, o algoritmo de ordenação pode executar alguns cálculos internos (que não estamos contando aqui) e, por fim, executará outra comparação entre dois outros elementos de S, que por sua vez também terá dois resultados possíveis. Sendo assim, podemos representar um algoritmo de ordenação baseado em comparação usando uma árvore de decisão T. Ou seja, cada nodo interno v de T corresponde a uma comparação, e as extremidades do nodo v' até seus filhos correspondem às computações que resultam em uma resposta "sim" ou em uma "não" (veja a Figura 4.14).

É importante observar que o algoritmo de ordenação hipotético em questão provavelmente não tenha conhecimento explícito da árvore T. Simplesmente usamos T para representar todas as seqüências possíveis de comparação que um algoritmo de ordenação pode fazer, iniciando na primeira comparação (associada com a raiz) e terminando na última comparação (associada com o pai de um nodo externo), antes do algoritmo terminar sua execução.

Cada ordem inicial possível ou ***permutação*** de elementos de S implica que nosso algoritmo hipotético de ordenação irá executar uma série de comparações, caminhando em T da raiz até algum nodo externo. Vamos associar com cada nodo externo v de T, então, o conjunto de permutações de S que fazem nosso algoritmo de ordenação encerrar em v. O ponto mais importante do nosso argumento relativo ao limite mínimo é que cada nodo externo v em T pode representar uma seqüência de comparações de, no máximo, uma permutação de S. A justificativa dessa afirmativa é simples: se duas permutações diferentes P_1 e P_2 de S são associadas com o mesmo nodo externo, então existem pelo menos dois objetos x_i e x_j de forma que x_i vem antes de x_j, em P_1, mas x_i vem depois de x_j em P_2. Ao mesmo tempo, a saída associada com v tem de ser uma reordenação específica de S, seja com x_i ou x_j aparecendo um antes do outro. Mas se tanto P_1 e P_2 fazem com que o algoritmo de ordenação resulte nos elementos de S nesta ordem, então isso implica que existe uma forma de forçar o algoritmo a fornecer um resultado onde x_i e x_j estão na ordem errada. Uma vez que isso não pode ser permitido por um algoritmo de ordenação correto, cada nodo externo de T deve ser associado com apenas uma permutação de S. Usamos essa propriedade da árvore de decisão associada com o algoritmo de ordenação para provar o seguinte resultado:

Teorema 4.12: *o tempo de execução de qualquer algoritmo de ordenação baseado em comparações é* $\Omega(n \log n)$ *para uma seqüência de n elementos, no pior caso.*

Prova: o tempo de execução de um algoritmo de ordenação baseado em comparação deve ser maior ou igual à altura da árvore de decisão T associada com este algoritmo, como descrito anteriormente (veja a Figura 4.14). Pelo argumento anterior, cada nodo externo T deve ser associado com uma permutação de S. Além disso, cada permutação de S deve resultar em um nodo externo diferente de T. O número de permutações de n objetos é

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1.$$

Sendo assim, T deve ter pelo menos $n!$ nodos externos. Pelo Teorema 2.8, a altura de T é pelo menos $\log(n!)$. Isso é suficiente para justificar o teorema, porque existem, no mínimo, $n/2$ termos que são maiores ou iguais a $n/2$ no produto $n!$; logo,

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\frac{n}{2},$$

que é $\Omega(n \log n)$. ■

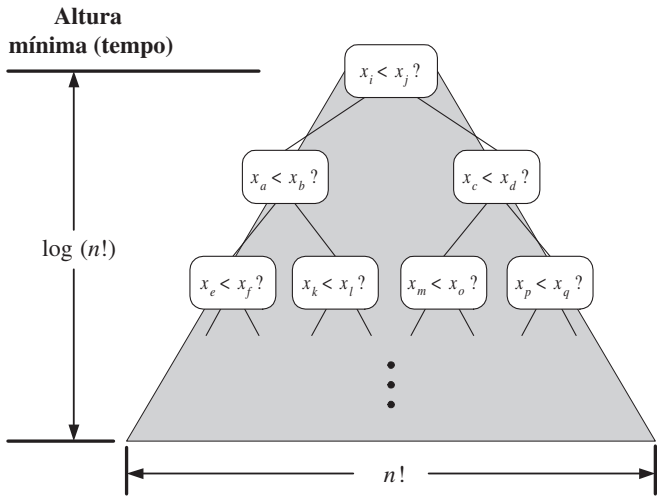


Figura 4.14 Visualização do limite mínimo para ordenação baseada em comparação.

4.5 *Bucket-sort* e *radix-sort*

Na seção anterior, mostramos que é necessário um tempo $\Omega(n \log n)$, no pior caso, para ordenar uma seqüência de n elementos usando um algoritmo de ordenação baseado em comparação. Uma questão natural, então, é se existem outros tipos de algoritmos de ordenação que podem ser projetados para executar assintoticamente de forma mais rápida que $O(n \log n)$. O interessante é que tais algoritmos existem; porém, requerem conhecimento de premissas especiais a respeito da seqüência a ser ordenada. Mesmo assim, tais cenários são freqüentes na prática, de maneira que discutir esses métodos vale a pena. Nesta seção, consideramos o problema de ordenar uma seqüência de itens, sendo cada um, um par de elementos-chave.

4.5.1 *Bucket-sort*

Consideremos uma seqüência S de n itens cujas chaves sejam inteiros no intervalo $[0, N - 1]$ para um inteiro $N \geq 2$ e suponhamos que S possa ser ordenado de acordo com as chaves dos itens. Neste caso, é possível ordenar S em tempo $O(n + N)$. Pode ser surpreendente, mas isto implica, por exemplo, que se N é $O(n)$, então podemos ordenar S em tempo $O(n)$. É claro que a questão principal é que, em razão da premissa restritiva sobre o formato dos elementos, evitemos as comparações.

A idéia é usar um algoritmo chamado ***bucket-sort***, que não é baseado em comparações, e sim no uso de chaves como índices em um arranjo de *buckets* B que tem entradas de 0 a $N - 1$. Um item com chave k é armazenado no "*bucket*" $B[k]$ que, por si só, é uma seqüência (de itens com chave k). Após inserir cada item da seqüência S em seu *bucket*, podemos colocar os itens de volta na seqüência S de forma ordenada pela enumeração do conteúdo dos *buckets* $B[0], B[1],..., B[N - 1]$, em ordem. Descrevemos o algoritmo *bucket-sort* no Algoritmo 4.15.

Algoritmo bucketSort(S):
  Entrada: seqüência S de itens com chaves inteiras no intervalo $[0, N-1]$
  Saída: seqüência S ordenada pelas chaves, de forma não-decrescente
  seja B um arranjo de N seqüências, cada uma inicialmente vazia
  para cada item x em S **faça**
    seja k ser a chave de x
    removemos x de S e o inserimos no fim do *bucket* (seqüência) $B[k]$
  para $i \leftarrow 0$ até $N-1$ **faça**
    para cada item x da seqüência $B[i]$ **faça**
      removemos x de $B[i]$ e o inserimos no fim de S

Algoritmo 4.15 *Bucket-sort.*

É fácil ver que o *bucket-sort* executa em tempo $O(n+N)$ e consome espaço $O(n+N)$, somente examinando os dois laços **para**.

Então, o *bucket-sort* é eficiente quando o intervalo de valores N para as chaves é pequeno se comparado à seqüência de tamanho n, digamos $N = O(n)$ ou $N = O(n \log n)$. Contudo, seu desempenho se deteriora à medida que N cresce comparado a n.

Uma propriedade importante do algoritmo *bucket-sort* é trabalhar corretamente mesmo se existem muitos elementos diferentes com a mesma chave. Na verdade, o descrevemos de forma a antecipar tais ocorrências.

Ordenação estável

Quando ordenando itens elemento-chave, uma questão importante é como as chaves iguais são tratadas. Seja $S = ((k_0, e_0),..., (k_{n-1}, e_{n-1}))$ uma seqüência de itens. Dizemos que um algoritmo de ordenação é ***estável*** se para quaisquer dois itens (k_i, e_i) e (k_j, e_j) de S, tal que $k_i = k_j$ e (k_i, e_i) precede (k_j, e_j) em S antes da ordenação (isto é, $i < j$), e o item (k_i, e_i) também precede (k_j, e_j), após a ordenação. A estabilidade é importante para um algoritmo de ordenação porque as aplicações podem querer preservar a ordenação inicial dos elementos com a mesma chave.

Nossa descrição informal do *bucket-sort* no Algoritmo 4.15 não garante estabilidade. Esta não é inerente ao método *bucket-sort* propriamente dito; porém, podemos facilmente modificar nossa descrição para tornar o *bucket-sort* estável, ao mesmo tempo que preservamos seu tempo de execução $O(n+N)$. Na verdade, podemos obter um algoritmo *bucket-sort* estável, removendo sempre ***o primeiro*** elemento da seqüência S e das seqüências $B[i]$, durante a execução do algoritmo.

4.5.2 *Radix-sort*

Uma das razões pela qual a estabilidade de um algoritmo é importante é que permite que a abordagem do *bucket-sort* seja aplicada a contextos mais gerais do que a ordenação de inteiros. Suponha, por exemplo, que queremos ordenar itens pares (k,l), onde k e l são inteiros no intervalo $[0, N-1]$ para qualquer inteiro $N \geq 2$. Em um contexto como esse, é natural definir a ordenação desses itens usando a convenção ***lexicográfica*** (do dicionário), onde $(k_1, l_1) < (k_2, l_2)$ se

- $k_1 < k_2$ ou
- $k_1 = k_2$ e $l_1 < l_2$.

Essa é uma versão aos pares da função lexicográfica de comparação, normalmente aplicada em *strings* de caracteres de mesmo tamanho (e facilmente generalizável para tuplas de d números com $d > 2$).

O algoritmo ***radix-sort*** ordena uma seqüência de pares tais como S, aplicando um *bucket-sort* estável sobre a seqüência duas vezes, primeiramente usando um componente do par como chave de ordenação e, em seguida, empregando o segundo componente. Mas qual é a ordem correta? Devemos ordenar primeiro pelos k (o primeiro componente) e em seguida nos l (o segundo componente) ou podemos fazê-lo de forma contrária?

Antes de respondermos a essa questão, consideremos o exemplo a seguir.

Exemplo 4.13: *consideremos a seguinte seqüência S:*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

Se ordenamos S de forma estável no primeiro componente, então obtemos a seqüência

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2).$$

Se, então, ordenamos de forma estável a seqüência S_1, usando o segundo componente, obtemos a seqüência

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7)),$$

que não é exatamente uma seqüência ordenada. Por outro lado, se ordenarmos S de forma estável usando o segundo componente, então obtemos a seguinte seqüência:

$$S_2 = ((1,2),(3,2), (2,2),(3,3), (2,3), (1,5),(2,5),(1,7)).$$

Se agora ordenarmos de forma estável a seqüência S_2,usando o primeiro componente, então obtemos a seqüência

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)),$$

que é, de fato, uma seqüência lexicograficamente ordenada.

Então, a partir desse exemplo, somos levados a acreditar que devemos primeiramente ordenar usando o segundo componente e, então, novamente usando o primeiro. Essa intuição é correta. Ordenando de forma estável, primeiramente pelo segundo componente e então novamente pelo primeiro, garantimos que, se dois elementos são iguais na segunda ordenação (pelo primeiro elemento), então sua ordem relativa na seqüência inicial (que é ordenada pelo segundo componente) é preservada. Sendo assim, a seqüência resultante tem a garantia de ser, todas as vezes, ordenada lexicograficamente. Deixamos para um exercício simples (R-4.15) a determinação sobre como essa abordagem pode ser estendida para triplas e outras d-tuplas de números. Podemos resumir esta seção como segue:

Teorema 4.14: *seja S uma seqüência de n itens elemento-chave, cada um dos quais tendo uma chave $(k_1,k_2,...,k_d)$, onde k_i é um inteiro no intervalo $[0, N-1]$ para qualquer inteiro $N \geq 2$. Podemos ordenar S lexicograficamente em tempo $O(d(n + N))$, usando radix-sort.*

Apesar de ser tão importante, a ordenação não é o único problema interessante que lida com relações de ordem total em um conjunto de elementos. Existem algumas aplicações, por exemplo, que não requerem a listagem ordenada de um conjunto inteiro, mas necessita, contudo, um montante de informação ordenada sobre o conjunto. Antes de estudarmos esse problema (chamado de "seleção"), vamos retroceder e brevemente comparar todos os algoritmos de ordenação estudados até agora.

4.6 Comparação dos algoritmos de ordenação

Neste ponto, pode ser útil uma interrupção para considerar todos os algoritmos que estudamos neste livro para ordenar uma seqüência de n elementos. Como muitas coisas na vida, não existe claramente o "melhor" algoritmo de ordenação, mas podemos oferecer algumas diretivas e observações baseadas nas propriedades conhecidas de "bons" algoritmos de ordenação.

Se bem implementado, o tempo de execução da *insertion-sort* é $O(n + k)$, onde k é o número de inversões (isto é, o número de pares de elementos fora de ordem). Sendo assim, a *insertion-sort* é um algoritmo excelente para ordenar pequenas seqüências (digamos com menos de 50 elementos), porque é simples de programar, e seqüências pequenas necessariamente contém poucas inversões. Além disso, *insertion-sort* é bastante eficiente para ordenar seqüências "quase" ordenadas. Por "quase", queremos dizer que o número de inversões é pequeno. Mas o desempenho $O(n^2)$ em relação ao tempo da *insertion-sort* torna-a uma escolha pobre fora dessas situações especiais.

Merge-sort, por outro lado, executa em tempo $O(n \log n)$, no pior caso, o que é ótimo para métodos de ordenação baseados em comparações. Além disso, estudos experimentais têm mostrado que, uma vez que é difícil fazer o *merge-sort* executar *in-place*, a carga de trabalho necessária para implementá-lo torna-o menos atrativo que as implementações *in-place* do *heapsort* e *quick-sort* para seqüências que caibam inteiras na memória principal do computador. Desta forma, *merge-sort* é um algoritmo excelente para situações em que a entrada não cabe toda na memória principal e tem de ser armazenada em blocos em um dispositivo de memória externa como um disco. Neste contexto, a forma que o *merge-sort* executa o processamento dos dados em grandes cadeias faz melhor uso de todos os dados trazidos em bloco do disco para a memória principal (veja a Seção 14.1.3).

Análises experimentais têm mostrado que, se a seqüência de entrada cabe inteiramente na memória principal, então as versões *in-place* do *quick-sort* e *heapsort* executam mais rapidamente que o *merge-sort*. De fato, o *quick-sort* tende, na média, a ser melhor do que o *heapsort*, nestes testes. Então, o ***quick-sort*** é uma escolha excelente como algoritmo de ordenação de finalidades genéricas no uso em memória. Na verdade, está incluído no utilitário qsort fornecido nas bibliotecas da linguagem C. Entretanto, seu desempenho temporal $O(n^2)$ para o pior caso faz do *quick-sort* uma escolha pobre para aplicações de tempo real em que temos de apresentar garantias sobre o tempo necessário para completar uma operação de ordenação.

Em cenários de tempo real, onde dispomos de um tempo fixo para executar uma operação de ordenação e os dados de entrada cabem na memória principal, o algoritmo ***heapsort*** provavelmente é a melhor escolha. Ele executa em tempo $O(n \log n)$ no pior caso e pode ser facilmente adaptado para executar *in-place*.

Finalmente, se nossa aplicação envolve ordenação por chaves inteiras ou d-tuplas de chaves inteira, então ***bucket-sort*** ou ***radix-sort*** são ótimas escolhas, pois executam em tempo $O(d(n + N))$ onde $[0, N - 1]$ é o intervalo de chaves inteiras (e $d = 1$ para o *bucket-sort*). Sendo assim, se $d(n + N)$ está "abaixo" de $n \log n$ (formalmente, $d(n + N)$ é $o(n \log n)$), então este método de ordenação pode executar mais rapidamente do que o *quick-sort* ou o *heapsort*.

Desta forma, nosso estudo sobre todos esses métodos diferentes provê nossa "caixa de ferramentas" para engenharia de algoritmos com uma coleção versátil de métodos de ordenação.

4.7 Seleção

Existe uma grande quantidade de aplicações na quais estamos interessados em identificar um único elemento em função de sua localização relativa à ordenação de um conjunto inteiro. Exemplos incluem a identificação do maior e do menor elementos, mas também podemos estar interessados em, digamos, identificar o termo ***mediano***, ou seja, o elemento tal que metade dos elementos sejam menores do que ele e a outra metade seja maior. Normalmente, consultas que questionam a respeito da localização de um elemento são chamadas de ***estatísticas de ordem***.

Nesta seção, vamos discutir o problema geral de estatística de ordem para selecionar o k-ésimo menor elemento de uma coleção não ordenada de n elementos comparáveis, o que é conhecido como o problema da ***seleção***. É claro, podemos resolver esse problema ordenando a coleção e então acessando a seqüência ordenada na localização $k - 1$.Usando o melhor dos algoritmos de ordenação baseado em comparação, essa abordagem irá levar tempo $O(n \log n)$. Logo, a questão que surge é se podemos obter um tempo de execução $O(n)$ para todos os valores de k, incluindo o interessante caso de encontrar o mediano, onde $k = \lfloor n/2 \rfloor$.

4.7.1 Poda e busca

Pode ser surpresa, mas, na verdade, podemos solucionar o problema da seleção em tempo $O(n)$ para qualquer valor de k. Além disso, a técnica que usamos para obter esse resultado envolve um interessante padrão de projeto de algoritmo, conhecido como ***poda-e-busca*** ou ***diminuição e conquista***. Aplicando esse padrão de projeto, resolvemos um problema que é definido a partir de uma coleção de n objetos podando uma fração destes e, recursivamente, resolvendo o problema menor. Quando tivermos finalmente reduzido o problema a um definido em uma coleção constante de objetos, então o resolvemos usando algum método de força bruta. Quando retornamos das chamadas recursivas, a construção se completa. Em alguns casos, podemos evitar o uso da recursão, caso em que simplesmente iteramos o passo de redução da poda e busca até que possamos aplicar um método de força bruta e parar.

4.7.2 *Quick-select* randomizado

Aplicando o padrão poda-e-busca ao problema de seleção, podemos projetar um método simples e prático chamado de ***quick-select randomizado*** para encontrar o k-ésimo menor elemento de um seqüência não ordenada de n elementos sobre os quais uma relação de ordem total é definida. O *quick-select* randomizado executa um tempo ***esperado*** $O(n)$, levando em conta todas as possíveis escolhas randômicas feitas pelo algoritmo – e esta expectativa não depende de qualquer premissa sobre a distribuição de entrada. Observamos que um *quick-select* randomizado executa em tempo $O(n^2)$, no ***pior caso***, fato cuja justificativa aparece no Exercício R-4.18. Também incluímos um exercício (C-4.24) que propõe modificar o *quick-select* para obter um algoritmo de seleção ***determinístico*** que execute em tempo $O(n)$, no ***pior caso***. Entretanto, a existência de um algoritmo determinístico é principalmente de interesse teórico, uma vez que o fator constante escondido pela notação O, neste caso, é relativamente grande.

Suponhamos uma dada seqüência não ordenada S de n elementos comparáveis juntamente com um inteiro $k \in [1, n]$. No nível superior, o algoritmo *quick-select* para encontrar o k-ésimo menor elemento de S é similar em estrutura ao algoritmo *quick-sort* randomizado descrito na Seção 4.3.1. Tomamos um elemento x de S randomicamente e o usamos como "pivô" para subdividir S em três subseqüências L, E e G, armazenando os elementos de S menores do que x, iguais a x e maiores do que x, respectivamente. Esta é a etapa de poda. Então, com base no valor de k, determinamos em quais destes conjuntos será aplicada a recursão. O *quick-select* randomizado é descrito no Algoritmo 4.16.

Algoritmo quickSelect(S,k):
  Entrada: seqüência S de n elementos comparáveis e um inteiro $k \in [1, n]$
  Saída: o k-ésimo menor elemento de S
  se $n = 1$ **então**
    retorne o (primeiro) elemento de S
  selecione um elemento aleatório x de S
  remova todos os elementos de S e coloque-os em três seqüências:

- L, armazenando os elementos de S menores do que x;
- E, armazenando os elementos de S iguais a x;
- G, armazenando os elementos de S maiores do que x.

  se $k \leq |L|$ **então**
    quickSelect(L,k)
  senão se $k \leq |L| + |E|$ **então**
    retorne x {cada elemento em E é igual a x}
  senão
    quickSelect(G,$k - |L| - |E|$) {observemos o novo parâmetro de seleção}

Algoritmo 4.16 Algoritmo *quick-select* randomizado.

4.7.3 Análise do *quick-select* randomizado

Mencionamos, anteriormente, que o algoritmo *quick-select* randomizado executa em tempo esperado $O(n)$. Felizmente, justificar essa afirmação requer apenas o mais sim-

ples argumento probabilístico. O principal fator probabilístico que usamos é o ***fator linear de expectativa***. Lembremos que esse fato define que, se X e Y são variáveis aleatórias e c é um número, então $E(X + Y) = E(X) + E(Y)$ e $E(cX) = cE(X)$, onde usamos $E(\mathcal{Z})$ para denotar o valor esperado para a expressão $\mathcal{Z}$.

Faça $t(n)$ indicar o tempo de execução do *quick-select* randomizado sobre uma seqüência de tamanho n. Uma vez que o algoritmo *quick-select* randomizado depende do resultado de eventos aleatórios, seu tempo de execução $t(n)$ é uma variável aleatória. Estamos interessados em limitar $E(t(n))$, o valor esperado de $t(n)$. Digamos que uma invocação recursiva do *quick-select* randomizado é "boa" se dividir S de maneira que o tamanho de L e G seja no máximo $3n/4$. Claramente, uma chamada recursiva é boa se tiver probabilidade de 1/2. Faça $g(n)$ indicar o número de invocações recursivas consecutivas (incluindo a presente), antes de obter uma boa invocação. Então

$$t(n) \leq bn \cdot g(n) + t(3n/4),$$

onde $b > 0$ é uma constante (para levar em conta o custo de cada chamada). Estamos, é claro, focando o caso em que n é maior que 1, pois podemos caracterizar de forma fechada que $t(1) = b$. Aplicando esta propriedade de expectativa linear ao caso geral, obtemos, então

$$E(t(n)) \leq E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + E(t(3n/4)).$$

Uma vez que uma chamada recursiva é considerada boa com probabilidade de 1/2 e que o fato de uma chamada recursiva ser boa ou não independe da chamada do pai ser boa, o valor esperado para $g(n)$ é o mesmo que o número esperado de vezes que temos de jogar uma moeda antes de obter "cara". Isso implica que $E(g(n)) = 2$. Conseqüentemente, se usarmos $T(n)$ como uma notação resumida para $E(t(n))$ (o tempo esperado de execução do algoritmo *quick-select* randomizado), então podemos escrever para o caso de $n > 1$

$$T(n) \leq T(3n/4) + 2bn.$$

Da mesma forma que para a equação de recorrência do *merge-sort*, gostaríamos de converter essa equação em uma forma fechada. Para fazer isso, vamos novamente aplicar iterativamente essa equação assumindo que n é grande. Então, por exemplo, após duas aplicações iterativas, obtemos

$$T(n) \leq T(3n/4^2)n) + 2b(3/4)n + 2bn.$$

Neste ponto, vemos que o caso geral é

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i.$$

Em outras palavras, o tempo de execução esperado para o *quick-select* randomizado é $2bn$ vezes a soma de uma progressão geométrica cuja base é um número positivo menor do que 1. Logo, pelo Teorema 1.12 sobre somatórios geométricos, obtemos como resultado que $T(n)$ é $O(n)$. Para resumir, temos:

Teorema 4.15: *o tempo de execução esperado para o quick-select randomizado em uma seqüência de tamanho n é $O(n)$.*

Como já mencionamos, existe uma variação do *quick-select* que não usa randomização e executa em tempo $O(n)$, no pior caso. O Exercício C-4.24 leva o leitor interessado ao projeto e análise deste algoritmo.

4.8 Exemplo em Java: *in-place quick-sort*

Lembremos da Seção 2.4.4 que um algoritmo de ordenação é dito ***in-place*** se usar apenas uma quantidade constante de memória além daquela necessária para os objetos em si sendo ordenados. O algoritmo *merge-sort*, como descrito anteriormente, não pode ser considerado *in-place*, e torná-lo *in-place* exige um algoritmo de junção mais complicado do que o discutido na Seção 4.1.1. Ordenação *in-place*, entretanto, não é difícil por natureza. Da mesma forma que o *heapsort*, o *quick-sort* pode ser adaptado para ser *in-place*.

Executar o algoritmo *quick-sort in-place* requer, entretanto, um pouco de ingenuidade, pois precisamos usar a própria seqüência de entrada para armazenar as subseqüências de todas as chamadas recursivas. Apresentamos o algoritmo inPlaceQuickSort que executa o *quick-sort in-place* no Algoritmo 4.17. O algoritmo inPlaceQuickSort considera que a seqüência de entrada S é composta por elementos distintos. A razão para essa restrição é explorada no Exercício R-4.12. A extensão para o caso geral é discutida no Exercício C-4.18. O algoritmo acessa os elementos da seqüência de entrada S, usando métodos baseados em colocações. Conseqüentemente, ele executa de forma eficiente desde que S seja implementado usando um arranjo.

```
Algoritmo inPlaceQuickSort(S, a, b):
    Entrada: seqüência S com elementos distintos; inteiros a e b
    Saída: seqüência S com elementos saídos das colocações a e b, inclusive, ordenados de forma não-decrescente da colocação a para b.
      se a ≥ b então retorne                {subintervalo vazio}
      p ← S.elemAtRank(b)                   {pivô}
      l ← a     {busca para a frente}
      r ← b – 1          {busca para trás}
      enquanto l ≤ r faça
        {encontra um elemento maior do que o pivô}
        enquanto l ≤ r e S.elemAtRank(l) ≤ p faça
          l ← l + 1
        {encontra um elemento menor do que o pivô }
        enquanto r ≥ l e S.elemAtRank(r) ≥ p faça
          r ← r – 1
        se l < r então
          S.swapElements(S.atRank(l),S.atRank(r))
      {colocamos o pivô no seu lugar definitivo}
      S.swapElements(S.atRank(l),S.atRank(b))
      {chamadas recursivas}
      inPlaceQuickSort(S, a, l – 1)
      inPlaceQuickSort(S, l + 1, b)
```

Algoritmo 4.17 *In-place quick-sort* para uma seqüência implementada sobre um arranjo.

O *in-place quick-sort* modifica a seqüência de entrada usando as operações swapElements e não cria subseqüências, explicitamente. Na verdade, uma subseqüência da seqüência de entrada é, implicitamente, representada por um intervalo de posições especificado pela colocação mais à esquerda *l* e pela colocação mais à direita *r*. O passo de divisão é executado pela varredura simultânea da seqüência tanto de *l* em diante como de *r* para trás, trocando os pares de elementos que se encontram na ordem contrária, como mostrado na Figura 4.18. Quando esses dois índices "se encontram", as subseqüências *L* e *G* estão em lados opostos do ponto de encontro. O algoritmo se completa ativando-se, recursivamente, sobre estas duas subsequências.

O *in-place quick-sort* reduz o tempo de execução necessário para a criação de novas seqüências e movimentação de elementos entre as mesmas por um fator constante. Apresentamos uma versão Java do *in-place quick-sort* no Trecho de Código 4.20.

Infelizmente, tecnicamente falando, a implementação de *quick-sort* apresentada não é realmente *in-place*, na medida em que requer mais do que uma quantidade constante de espaço adicional. É claro, não estamos usando espaço extra para as subseqüências e estamos utilizando apenas uma quantidade constante de espaço adicional para variáveis locais (tais como *l* e *r*). Então, de onde vem o espaço adicional? Ele vem da recursão, pois, lembrando a Seção 2.1.1, observamos que, para o *quick-sort*, necessitamos espaço para a pilha proporcional à profundidade da árvore de recursão, que é no mínimo log *n* e no máximo *n* – 1. De maneira a tornar o *quick-sort* verdadeiramente *in-place*, precisamos implementá-lo de forma não-recursiva (e não usar a pilha). A chave para tal implementação é que necessitamos uma forma *in-place* de determinar os limites para os extremos direito e esquerdo da subseqüência "corrente". Tal esquema não é difícil, e deixamos os detalhes desta implementação para o Exercício (C-4.7).

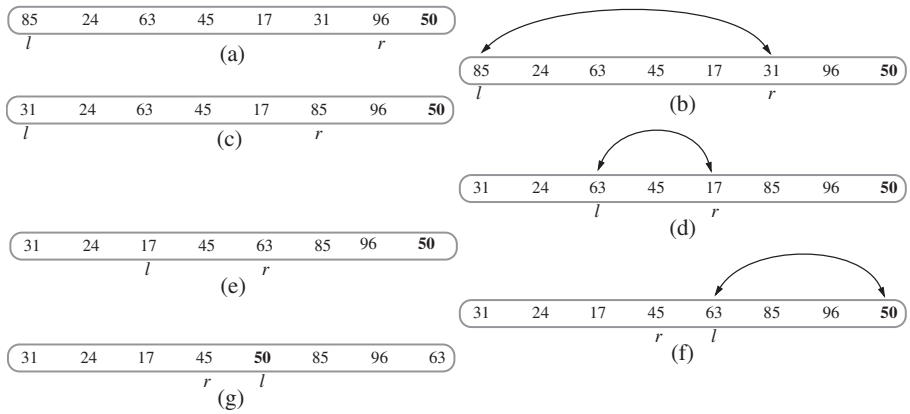


Figura 4.18 Passo de divisão do *in-place quick-sort*. O índice *l* varre a seqüência da esquerda para direita, e o índice *r* varre a seqüência da direita para a esquerda. Uma troca é efetuada quando *l* está em um elemento maior do que o pivô, e *r* em um elemento menor do que o pivô. Uma troca final com o pivô completa o passo de divisão.

```
/**
 * Ordenamos os elementos da seqüência S em ordem não-decrescente de acordo
 * com o comparador c, usando o algoritmo quick-sort. A maior parte do trabalho
 * é feito pelo método auxiliar recursivo quickSortStep.
/**
public static void quickSort (Sequence S, Comparator c) {
    if (S.size() < 2)
      return; // uma seqüência com 0 ou 1 elemento é considerada ordenada
    quickSortStep(S, c, 0, S.size() –1); // método recursivo de ordenação
}
/**
 * Ordenamos em ordem não-decrescente os elementos da seqüência S entre
 * as colocações leftBound e rightBound, usando uma implementação in-place
 * recursiva do algoritmo quick-sort.
**/
private static void quickSortStep (Sequence S, Comparator c,
                                   int leftBound, int rightBound ) {
    if (leftBound >= rightBound)
      return;
    Object pivot = S.atRank(rightBound).element();
    int leftIndex = leftBound; // vai varrer para a direita
    int rightIndex = rightBound–1; // vai varrer para a esquerda
    while (leftIndex <= rightIndex) {
      // varre para a direita buscando um elemento maior que o pivô
      while ( (leftIndex <= rightIndex) &&
              c.isLessThanOrEqualTo(S.atRank(leftIndex).element(), pivot) )
              leftIndex++;
      // varre para a esquerda buscando um elemento menor do que o pivô
      while ( (rightIndex >= leftIndex) &&
              c. isGreaterThanOrEqualTo(S.atRank(rightIndex).element(), pivot) )
                rightIndex– –;
      if (leftIndex < rightIndex) // ambos os elementos foram encontrados
        S.swapElements(S.atRank(leftIndex), S.atRank(rightIndex));
  } //o laço continua até os índices se cruzarem
  // posicionamos o pivô trocando-o pelo elemento localizado em leftIndex
    S.swapElements(S.atRank(leftIndex), S.atRank(rightBound));
  // o pivô agora esta localizado em leftIndex, então o aplicamos, recursivamente,
  nos dois lados do mesmo
  quickSortStep(S, c, leftBound, leftIndex –1);
  quickSortStep(S, c, leftIndex+1, rightBound);
}
```

Trecho de Código 4.20 Implementação Java do *in-place quick-sort*. Assumimos que a seqüência de entrada é implementada com um arranjo que armazena elementos distintos.

4.9 Exercícios

Reforço

R-4.1 Forneça uma justificativa completa para o Teorema 4.1.

R-4.2 Forneça uma descrição em pseudocódigo do algoritmo *merge-sort*. Você pode chamar o algoritmo merge como uma sub-rotina.

R-4.3 Forneça uma descrição em pseudocódigo de uma variação do algoritmo *merge-sort* que use um arranjo no lugar de uma seqüência generalizada. ***Dica***: use um arranjo como "*buffer*".

R-4.4 Demonstre que o tempo de execução do algoritmo *merge-sort* em uma seqüência de n elementos é $O(n \log(n))$ mesmo quando n não é uma potência de 2.

R-4.5 Suponha que temos duas seqüências ordenadas A e B com n elementos cada e que elas não devem ser interpretadas como conjuntos (ou seja, A e B podem conter entradas duplicadas). Descreva um método de tempo $O(n)$ para determinar uma seqüência representando o conjunto $A \cup B$ (sem duplicatas).

R-4.6 Demonstre que $(X - A) \cup (X - B) = X - (A \cap B)$ para quaisquer conjuntos X, A e B.

R-4.7 Suponha que implementamos uma partição de uma estrutura de dados baseada em árvore (*union* e *find*) usando somente a heurística "união por tamanho". Qual é o tempo de execução amortizado de uma seqüência de n operações union e find neste caso?

R-4.8 Forneça descrições em pseudocódigo para realizar os métodos insert e remove em um conjunto implementado com uma seqüência ordenada.

R-4.9 Suponha que modificamos a versão determinística do algoritmo *quick-sort* de forma que, em vez de selecionar como pivô o último elemento em uma seqüência de n elementos, escolhemos o elemento na posição (índice) $\lfloor n/2 \rfloor$, isto é, um elemento no meio da seqüência. Qual o tempo de execução desta versão do *quick-sort* em uma seqüência já ordenada?

R-4.10 Considere mais uma vez a versão modificada da versão determinística do algoritmo *quick-sort* de forma que, em vez de selecionar como pivô o último elemento em uma seqüência de n elementos, escolhemos o elemento na posição $\lfloor n/2 \rfloor$. Descreva o tipo de seqüência que faria com que esta versão do *quick-sort* fosse executada em tempo $\Theta(n^2)$.

R-4.11 Mostre que o tempo de execução de melhor caso do *quick-sort* em uma seqüência de tamanho n com elementos distintos é $O(n \log(n))$.

R-4.12 Suponha que o algoritmo inPlaceQuickSort (Algoritmo 4.17) é executado em uma seqüência com elementos duplicados. Mostre que, neste caso, o algoritmo ordena corretamente a seqüência de entrada, mas o resultado do passo de divisão pode diferir da descrição de alto nível dada na Seção 4.3 e resultar em ineficiência. O que acontece no passo de partição quando existem elementos iguais ao pivô? A seqüência E (armazenando os elementos iguais ao pivô) é realmente avaliada? O algoritmo usa as subseqüências L e R ou outras subseqüências? Qual o tempo de execução do algoritmo se todos os elementos da entrada forem iguais?

R-4.13 Forneça uma descrição em pseudocódigo da versão *in-place* do *quick-sort* especializada para receber um arranjo como entrada em vez de uma seqüência generalizada, e retornando o mesmo arranjo como saída.

R-4.14 Quais dos algoritmos de *bubble-sort*, *heapsort*, *merge-sort* e *quick-sort* são estáveis?

R-4.15 Descreva um método de *radix-sort* para ordenar lexicograficamente uma seqüência S de triplas (k, l, m) onde k, l e m são inteiros no intervalo $[0, N - 1]$ para algum $N \geq 2$. Como o método pode ser estendido para seqüências de d-tuplas $(k_1, k_2,..., k_d)$, onde cada k_i é um inteiro no intervalo $[0, N - 1]$?

R-4.16 O algoritmo *bucket-sort* é *in-place*? Justifique.

R-4.17 Forneça uma descrição em pseudocódigo de um algoritmo *in-place quick-select*.

R-4.18 Demonstre que o tempo de execução de pior caso do algoritmo de *quick-select* em uma seqüência de n elementos é $\Omega(n^2)$.

Criatividade

C-4.1 Mostre como implementar o método equals(B) em um conjunto A, que testa quando $A = B$, em tempo $O(|A| + |B|)$, através de uma variação do algoritmo merge genérico e assumindo que A e B são implementados com seqüências ordenadas.

C-4.2 Forneça uma variação do algoritmo merge genérico para determinar $A \oplus B$, que é o conjunto de elementos que estão em A ou em B, mas não em ambos.

C-4.3 Suponha que implementamos o TAD conjunto representando cada conjunto com uma árvore balanceada de pesquisa. Descreva e analise algoritmos para cada um dos métodos do TAD conjunto.

C-4.4 Seja A uma coleção de objetos. Descreva um método eficiente para converter A em um conjunto, ou seja, remover todas as repetições em A. Qual o tempo de execução de seu algoritmo?

C-4.5 Considere conjuntos cujos elementos são (ou podem ser mapeados para) inteiros no intervalo $[0, N - 1]$. Uma forma popular de representar um conjun-

to A deste tipo é através de um vetor booleano B onde dizemos que x está em A se e somente se $B[x]$ = true. Como cada posição de B pode ser representada por um *bit*, B é às vezes chamado de ***vetor de bits***. Descreva algoritmos eficientes para realizar os métodos union, intersection e subtraction para o TAD conjunto assumindo essa representação. Quais são os tempos de execução desses métodos?

C-4.6 Suponha que implementemos uma estrutura de dados de partição baseada em árvore (*union* e *find*) usando a heurística da união pelo tamanho e uma herística de compressão de caminhos ***parcial***. A compressão parcial de caminhos significa, neste caso, que após executar uma seqüência de passagens por ponteiros para uma operação find, atualizamos o ponteiro pai para cada nodo u ao longo deste caminho para apontar para seu avô. Mostre que o tempo total de execução para executar n operações union e find ainda é $O(n \log^* n)$.

C-4.7 Suponha que implementemos uma estrutura de dados de partição baseada em árvore (*union* e *find*) usando as heurísticas união pelo tamanho e compressão de caminhos. Demonstre que o tempo total para executar n operações union e find é $O(n)$ se todas as union ocorrerem antes de todas as find.

C-4.8 Demonstre que o *quick-sort* randomizado é executado em tempo $O(n \log(n))$ com probabilidade $1 - 1/n^2$.
Dica: use o ***limite de Chernoff***, que diz que, se jogarmos uma moeda k vezes, então a probabilidade de obtermos menos de $k/16$ caras é menor do que $2^{-k/8}$.

C-4.9 Suponha que temos uma seqüência S de n elementos, cada um deles pintado de azul ou de vermelho. Suponha que S é representado por um arranjo e forneça um método *in-place* para ordenar S de forma que todos os elementos azuis sejam listados antes de todos os elementos vermelhos. Você pode estender sua abordagem para três cores?

C-4.10 Suponha que temos uma seqüência S de n elementos, de forma que cada elemento em S represente um voto diferente numa eleição, onde cada voto é dado como um inteiro representando a identificação do candidato escolhido. Sem assumir hipóteses sobre os candidatos ou o número de candidatos, planeje um algoritmo de tempo $O(n \log(n))$ para determinar quem vence a votação representada por S, assumindo que o candidato com o maior número de votos será o escolhido.

C-4.11 Considere o problema de votação do exercício anterior, mas agora suponha que conhecemos o número $k < n$ de candidatos. Descreva um algoritmo de tempo $O(n \log(k))$ para determinar quem vence a eleição.

C-4.12 Demonstre que qualquer algoritmo de ordenação baseado em comparações pode ser transformado em um método estável sem afetar o tempo de execução assintótico do algoritmo.
Dica: altere a forma de comparar os elementos uns com os outros.

C-4.13 Suponha que temos duas seqüências A e B de n elementos cada, possivelmente contendo repetições, onde existe uma relação de ordem total. Descre-

va um algoritmo eficiente para determinar se A e B contêm o mesmo conjunto de elementos (possivelmente em ordens diferentes). Qual o tempo de execução deste método?

C-4.14 Suponha que temos uma seqüência S de n elementos, cada um deles um inteiro no intervalo $[0, n^2 - 1]$. Descreva um método simples para ordenar S em tempo $O(n)$.
Dica: pense em formas alternativas de interpretar os elementos.

C-4.15 Sejam $S_1, S_2,..., S_k$ k seqüências diferentes cujos elementos têm chaves inteiras no intervalo $[0, N - 1]$ para algum $N \geq 2$. Descreva um algoritmo com tempo $O(n + N)$ para ordenar todas as seqüências (sem fazer sua união), onde n denota o tamanho total de todas as seqüências.

C-4.16 Suponha que temos uma seqüência S de n elementos onde existe uma relação de ordem total. Descreva um método eficiente para determinar se existem dois elementos iguais em S. Qual o tempo de execução de seu algoritmo?

C-4.17 Descreva uma versão não-recursiva e *in-place* do algoritmo *quick-sort*. O algoritmo deve ser baseado na mesma estratégia de divisão e conquista.
Dica: pense em como "marcar" os extremos da esquerda e da direita da subseqüência corrente antes de fazer uma chamada recursiva.

C-4.18 Modifique o algoritmo inPlaceQuickSort (Algoritmo 4.17) para tratar eficientemente o caso geral em que a seqüência de entrada S pode ter chaves repetidas.

C-4.19 Seja S uma seqüência de n elementos onde uma relação de ordem total esteja definida. Uma inversão em S é um par de elementos x e y tais que x aparece antes de y em S, mas $x > y$. Descreva um algoritmo de tempo $O(n \log(n))$ para determinar o número de inversões em S.
Dica: tente modificar o algoritmo *merge-sort* para resolver este problema.

C-4.20 Seja S uma seqüência de n elementos onde uma relação de ordem total esteja definida. Descreva um algoritmo baseado em comparações para ordenar S em tempo $O(n + k)$, onde k é o número de inversões em S (veja a definição de inversão no problema anterior).
Dica: pense em uma versão *in-place* do algoritmo *insertion-sort* que, depois de um passo linear de pré-processamento apenas troca de lugar elementos que estão invertidos.

C-4.21 Apresente uma seqüência de n inteiros com (n^2) inversões. (Veja a definição de inversão no Exercício C-4.19.)

C-4.22 Sejam A e B duas seqüências de n inteiros cada. Dado um inteiro x, descreva um algoritmo de tempo $O(n \log(n))$ para determinar se há um inteiro a em A e um inteiro b em B, tais que $x = a + b$.

C-4.23 Dada uma seqüência S de n elementos comparáveis, descreva um método eficiente para achar os $\lceil \sqrt{n} \rceil$ itens cuja posição em numa versão ordenada de S é a mais próxima da mediana. Qual o tempo de execução do seu método?

C-4.24 Este problema trata de uma modificação do algoritmo de *quick-select* para torná-lo determinístico e ainda podendo ser executado em tempo $O(n)$ em uma seqüência de n elementos. A idéia é modificar a forma com que determinamos o pivô de modo a ser escolhido deterministicamente, e não aleatoriamente, como segue:

> Divida o conjunto S em $\lceil n/5 \rceil$ grupos de tamanho 5 cada (exceto para talvez um grupo). Ordene cada pequeno conjunto e identifique a mediana do conjunto. Com estas $\lceil n/5 \rceil$ medianas "pequenas", aplique o algoritmo de seleção recursivamente para encontrar a mediana das medianas "pequenas". Use esse elemento como pivô e continue como no algoritmo *quick-select*.

Demonstre que esse método determinístico tem tempo $O(n)$, respondendo às seguintes questões (ignore funções "piso" e "teto" se isso simplificar a matemática, pois os resultados assintóticos continuam os mesmos):

a. Quantas medianas "pequenas" são menores ou iguais ao pivô escolhido? Quantas são maiores ou iguais?
b. Para cada mediana "pequena" menor ou igual ao pivô, quantos elementos são menores ou iguais ao pivô? O mesmo se aplica àquelas maiores ou iguais ao pivô?
c. Discuta por que o método para encontrar o pivô deterministicamente e usá-lo para particionar S leva tempo $O(n)$.
d. Com base nessas estimativas, escreva uma equação de recorrência que limita o tempo de execução de pior caso $t(n)$ para este algoritmo de seleção (note que no pior caso há duas chamadas recursivas – uma para encontrar a mediana das medianas "pequenas" e outra para fazer a recorrência com o maior valor entre L e G).
e. Usando essa relação de recorrência, mostre por indução que $t(n)$ é $O(n)$.

C-4.25 Bob tem um conjunto A de n porcas e um conjunto B de n parafusos, de forma que cada porca em A tem um único parafuso correspondente em B. Infelizmente, todas as porcas em A se parecem, e todos os parafusos em B também. O único tipo de comparação que Bob pode fazer é pegar um par (a, b) de porca e parafuso sendo $a \in A$, $b \in B$ e testar se a é maior, menor ou encaixa em b. Descreva um algoritmo eficiente para Bob encontrar todos os pares de porcas e parafusos. Qual o tempo de execução desse algoritmo, em função do número de testes que Bob terá de fazer?

C-4.26 Demonstre como um algoritmo de seleção de tempo $O(n)$ determinístico pode ser usado para projetar um algoritmo de ordenação do tipo *quick-sort* que executa em tempo $O(n \log n)$ no ***pior caso*** em uma seqüência de n elementos.

C-4.27 Dada uma seqüência não-ordenada S de n elementos comparáveis e um inteiro k, mostre um algoritmo de tempo esperado $O(n \log k)$ para encontrar os elementos $O(k)$ que estão na colocação $[n/k]$, $2[n/k]$, $3[n/k]$, e assim por diante.

Projetos

P-4.1 Projete e implemente uma versão estável do algoritmo *bucket-sort* para ordenar uma seqüência de n elementos com chaves inteiras no intervalo $[0, N - 1]$ para $N \geq 2$. O algoritmo deve ser executado em tempo $O(n + N)$. Realize testes para verificar se esse método realmente é executado no tempo proposto para vários valores de n e N e faça um resumo descrevendo o código e os resultados dos testes.

P-4.2 Implemente o *merge-sort* e o *quick-sort* determinístico e realize testes para verificar qual dos dois é mais rápido. Seus testes devem incluir seqüências que aparentemente sejam "aleatórias'' e seqüências que pareçam "quase" ordenadas ou inversamente ordenadas. Redija um resumo descrevendo o código e os resultados dos testes.

P-4.3 Implemente o *quick-sort* determinístico e sua versão randomizada e realize testes para verificar qual dos dois é mais rápido. Seus testes devem incluir seqüências que aparentemente sejam "aleatórias" e seqüências que pareçam "quase" ordenadas. Em um resumo, descreva o código e os resultados dos testes.

P-4.4 Implemente uma versão *in-place* do *insertion-sort* e uma versão *in-place* do *quick-sort*. Realize testes para determinar os valores de n para os quais o *quick-sort* é mais rápido (em média) do que o *insertion-sort*.

P-4.5 Projete e implemente uma animação para um dos algoritmos de ordenação apresentados neste capítulo. Sua animação deve ilustrar as propriedades essenciais do algoritmo de forma intuitiva e deve ter texto ou som para explicar o algoritmo para alguém que não o conheça. Em um resumo, descreva a animação.

P-4.6 Implemente o TAD partição (*union* e *find*) usando a abordagem baseada em árvore com as heurísticas união pelo tamanho e compressão de caminhos.

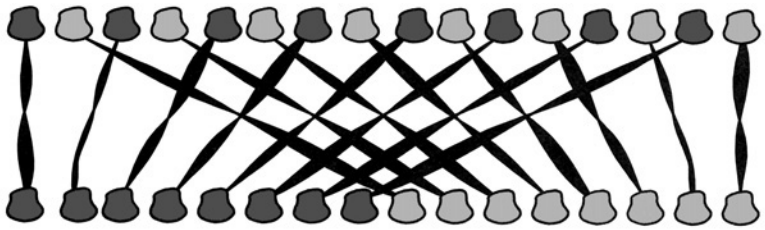
Notas

O clássico texto de Knuth *Sorting and Searching* [119] contém uma extensa história do problema da classificação e algoritmos para resolvê-lo, começando com as máquinas de classificação baseadas em cartões para realizar o censo no século XIX. Huang e Langston [103] descrevem como unificar duas listas ordenadas de uma forma *in-place* e com tempo linear. Nosso TAD conjunto é derivado do TAD conjunto de Aho, Hopcroft e Ullman [8]. O algoritmo padrão para *quick-sort* é atribuído a Hoare [96]. Uma análise mais profunda do *quick-sort* randomizado pode ser encontrada no livro de Motwani e Raghavan [157]. Gonnet e Baeza-Yates [81] fornecem comparações experimentais e análises teóricas de uma série de algoritmos diferentes de ordenação. O termo "poda e busca" é originário da literatura de geometria computacional (como no livro de Clarkson [47] e Meggido [145,146]). O termo "diminuição e conquista'' é de Levitin [132].

A análise que fornecemos para a estrutura de dados de partição vem de Hopcroft e Ullmann [99] (veja também [7]). Tarjan [199] mostra que uma seqüência de n operações union e find implementadas como descritas neste capítulo podem ser executadas em tempo $O(n\alpha(n))$, onde $\alpha(n)$ é a função inversa da função de Ackermann que tem crescimento muito lento e limite apertado no pior caso (veja também [200]). Gabow e Tarjan [73] mostram, entretanto, que se pode obter, em alguns casos, um tempo de execução $O(n)$.

Capítulo

5 Técnicas fundamentais



5.1 O método guloso 264

5.1.1 O problema da mochila fracionária 264

5.1.2 Escalonamento de tarefas 265

5.2 Divisão e conquista 267

5.2.1 Equações recursivas de divisão e conquista 268

5.2.2 Multiplicação de inteiros 274

5.2.3 Multiplicação de matrizes 276

5.3 Programação dinâmica 277

5.3.1 Produto de matrizes encadeadas 278

5.3.2 A técnica genérica 281

5.3.3 O problema da mochila 0-1 282

5.4 Exercícios 285

Uma rede popular de televisão transmite dois programas sobre carpintaria. Em um dos programas, o apresentador constrói móveis usando ferramentas específicas, enquanto que no outro, o apresentador constrói móveis usando ferramentas comuns. As ferramentas especializadas do primeiro programa são muito boas para as tarefas para as quais foram projetadas, mas não são muito versáteis. As ferramentas do segundo programa, entretanto, são fundamentais, porque podem ser usadas em uma grande variedade de tarefas diferentes.

Esses dois programas de televisão nos apresentam uma metáfora interessante a respeito de estruturas de dados e projeto de algoritmos. Existem alguns algoritmos altamente especializados. São eficientes para os problemas que se propõem a resolver, mas não são muito versáteis. Existem outros algoritmos, entretanto, que são ***fundamentais***, porque podem ser aplicados em uma grande gama de problemas de projeto de estruturas de dados e algoritmos. Aprender a usar essas técnicas fundamentais é uma arte, e este capítulo é dedicado a desenvolver o conhecimento necessário para usar essas técnicas efetivamente.

As técnicas fundamentais apresentadas neste capítulo são o método guloso, divisão e conquista e programação dinâmica. Essas técnicas são versáteis, e exemplos são apresentados neste e nos demais capítulos deste livro.

O método guloso é usado nos algoritmos de grafos balanceados discutidos no Capítulo 7, bem como no problema de compressão de dados apresentado na Seção 9.3. A idéia principal desta técnica, como o próprio nome diz, é fazer uma série de escolhas gulosas de forma a construir uma solução ótima (ou uma solução próxima do ótimo) para um dado problema. Neste capítulo, apresentaremos a estrutura genérica do método guloso e mostraremos como ele pode ser aplicado nos problemas da mochila e de escalonamento.

Divisão e conquista é usado nos algoritmos *merge-sort* e *quick-sort* do Capítulo 4. A idéia principal desta técnica é resolver um problema dividindo-o em problemas menores recursivamente, até que sejam pequenos o suficiente para ser solucionados por força bruta e, à medida que as chamadas recursivas retornam, juntar todos os subproblemas para obter a solução do problema original. Neste capítulo, mostraremos como projetar e analisar algoritmos genéricos de divisão e conquista e forneceremos aplicações adicionais desta técnica para os problemas de multiplicação de inteiros e matrizes grandes. Apresentaremos, também, algumas técnicas para resolver equações recorrentes por divisão e conquista, incluindo um teorema genérico que pode ser aplicado a uma grande variedade de equações.

A técnica de programação dinâmica inicialmente aparenta ser misteriosa, mas é extremamente poderosa. A idéia principal é resolver um problema caracterizando seus subproblemas usando um pequeno conjunto de índices inteiros. O objetivo desta caracterização é permitir que uma solução ótima para o problema seja definida por uma combinação de soluções (possivelmente sobrepostas) para problemas menores. Se podemos construir tal caracterização, que é o passo mais difícil da programação dinâmica, então podemos construir um algoritmo direto que resolve grandes problemas a partir de problemas menores. Essa técnica está por trás do algoritmo de fechamento transitivo de Floyd-Warshall apresentado no Capítulo 6. Neste capítulo, descreveremos um *framework* genérico para programação dinâmica e forneceremos várias aplicações, incluindo o problema da mochila 0-1.

5.1 O método guloso

A primeira técnica que consideraremos neste capítulo é o ***método guloso***. Caracterizamos esse padrão de projeto em termos da propriedade genérica da ***escolha gulosa***, e apresentaremos duas aplicações de seu uso.

O método guloso é aplicado em problemas de otimização, isto é, problemas que envolvem pesquisa em um conjunto de ***configurações*** para encontrar uma que minimize ou maximize uma ***função objetivo*** definida para essas configurações. A fórmula genérica do método guloso não poderia ser mais simples. Para resolver um problema de otimização, fazemos uma seqüência de escolhas. Essa seqüência se inicia a partir de uma configuração bem-entendida e então, iterativamente, escolhe a que parece ser a melhor do conjunto das possíveis.

A abordagem gulosa nem sempre tende para a solução ótima. Existem vários problemas, entretanto, em que funciona bem, e tais problemas são os ditos possuidores da propriedade da ***escolha gulosa***. Essa propriedade diz que a solução global ótima é alcançada a partir de um conjunto de escolhas locais ótimas (isto é, escolhas que são as melhores dentre um conjunto de possibilidades disponíveis no momento), começando por uma configuração bem-definida.

5.1.1 O problema da mochila fracionária

Consideremos o problema da ***mochila fracionária***, em que fornecemos um conjunto S de n itens de maneira que cada item i tenha um benefício positivo b_i e um peso positivo w_i, e que queiramos encontrar o conjunto que nos traga o maior benefício sem exceder um peso determinado W. Se estivermos restritos a poder aceitar ou recusar cada item, então teremos a versão 0-1 deste problema (para o qual fornecemos uma solução usando programação dinâmica da Seção 5.3.3). Vamos então nos permitir pegar pedaços arbitrários de alguns elementos. A motivação do problema da mochila fracionária é estarmos indo em uma viagem e dispormos de uma única mochila que pode carregar itens que juntos não pesem mais de W. Além disso, podemos quebrar os itens em pedaços, arbitrariamente. Assim, podemos pegar uma certa porção x_i de cada item i de forma que

$$0 \le x_i \le w_i \text{ para cada } i \in S \quad \text{e} \quad \sum_{i \in S} x_i \le \mathrm{W}.$$

O benefício total do conjunto de itens é determinado pela função objetiva

$$\sum_{i \in S} b_i (x_i / w_i).$$

Consideremos, por exemplo, um estudante que está indo para um evento esportivo ao ar livre e deve encher sua mochila com generos alimentícios suficientes para levar. Cada alimento é algo que pode ser facilmente dividido em frações, tais como refrigerantes, salgadinhos, chocolates e bolachas.

Algoritmo FractionalKnapsack(S,W):

Entrada: conjunto S de itens onde cada item $i \in S$ tem um benefício positivo b_i e um peso positivo w_i; peso máximo total positivo W

Saída: quantidade x_i de cada item $i \in S$ que maximiza o
benefício total sem exceder o peso máximo total W
para cada item $i \in S$ **faça**
$x_i \leftarrow 0$
$v_i \leftarrow b_i / w_i$ {***valor do índice*** do item i}
$w \leftarrow 0$ {peso total}
enquanto $w < W$ **faça**
removemos de S o item i com o maior valor de índice {escolha gulosa}
$a \leftarrow \min\{ w_i, W - w\}$ {mais que $W - w$ implica excesso de peso}
$x_i \leftarrow a$
$w \leftarrow w + a$

Algoritmo 5.1 Algoritmo guloso para o problema da mochila fracionária.

Essa é uma situação em que a gula é boa, pois resolvemos o problema da mochila fracionária usando a abordagem gulosa mostrada no Algoritmo 5.1.

O algoritmo FractionalKnapsack pode ser implementado em tempo $O(n \log n)$, onde n é a quantidade de itens em S. Em particular, usamos uma fila de prioridades baseada em um *heap* (Seção 2.4.3) para armazenar os itens de S, onde a chave de cada item é o valor do seu índice. Usando esta estrutura de dados, cada escolha gulosa que remove o item de maior valor para o índice consome tempo $O(\log n)$.

Para verificar se o problema da mochila fracionária satisfaz à propriedade da escolha gulosa, suponha que existem dois itens i e j, tais que

$$x_i < w_i, \quad x_j > 0 \quad \text{e} \quad v_i > v_j.$$

Façamos

$$y = \min\{w_i - x_i, x_j\}.$$

Podemos substituir, então, uma certa quantidade y do item j por uma quantidade igual do item i aumentando o benefício sem aumentar o peso total. Assim, podemos calcular corretamente as quantidades ótimas de cada item escolhendo "com gula" os itens com maior índice de valor. Essa propriedade é determinada pelo seguinte teorema:

Teorema 5.1: *dada uma coleção S de n itens, de maneira que cada item tenha um benefício b_i e peso w_i, podemos construir um subconjunto de S com benefício máximo, permitindo quantidades fracionadas que têm peso W em tempo $O(n \log n)$.*

Esse teorema mostra como podemos resolver de forma eficiente a versão fracionada do problema da mochila. A versão tudo-ou-nada ou "0-1" do problema da mochila, entretanto, não satisfaz à propriedade da escolha gulosa, e resolver esta versão do problema é muito mais difícil, como verificaremos nas Seções 5.3.3 e 13.3.4.

5.1.2 Escalonamento de tarefas

Consideremos outro problema de otimização. Suponha que temos um conjunto T de ***tarefas***, tal que cada tarefa i tem um ***tempo inicial*** s_i e um tempo final f_i (onde $s_i < f_i$). A tarefa i deve iniciar no tempo s_i, e terminar no tempo f_i. Cada tarefa deve ser executada em uma ***máquina***, e cada máquina executa apenas uma tarefa por vez. Duas tarefas i e j são **não-conflitantes** se $f_i \le s_j$ ou $f_j \le s_i$. Duas tarefas podem ser escalonadas para ser executadas na mesma máquina apenas se forem não-conflitantes.

O problema de ***escalonamento de tarefas*** que consideramos aqui é escalonar todas as tarefas de T no menor número de máquinas possível, evitando os conflitos. Por outro lado, podemos imaginar as tarefas como reuniões que devemos organizar no menor número de salas possível (veja a Figura 5.2).

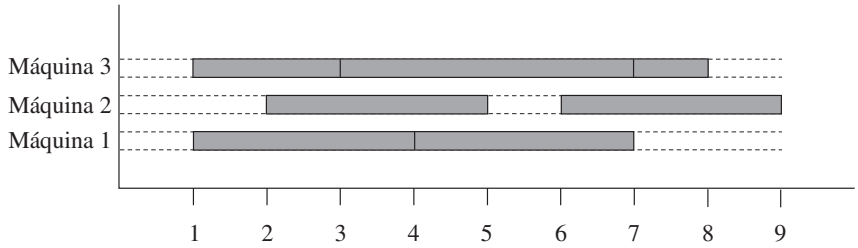


Figura 5.2 Representação de uma solução para o problema de escalonamento de tarefas cuja coleção de pares de tempos inicial e final é {(1,3), (1,4), (2,5), (3,7), (4,7), (6,9), (7,8)}.

No Algoritmo 5.3 descrevemos um algoritmo guloso simples para esse problema.

Algoritmo TaskSchedule(T):
  Entrada: um conjunto T de tarefas onde cada tarefa tem um tempo inicial s_i e um tempo final f_i
  Saída: um escalonamento não-conflitante para as tarefas de T usando uma quantidade mínima de máquinas
  $m \leftarrow 0$ {número ótimo de máquinas}
  enquanto $T \neq \emptyset$ **faça**
    remova de T a tarefa i com menor tempo inicial s_i
    se existe uma máquina j sem tarefas conflitantes com a tarefa i **então**
      programamos a tarefa i para a máquina j
    senão
      $m \leftarrow m + 1$ {acrescente uma nova máquina}
      programamos a tarefa i para a máquina m

Algoritmo 5.3 Algoritmo guloso para o problema do escalonamento de tarefas.

Correção do escalonamento de tarefas guloso

No algortimo TaskSchedule, começamos sem nenhuma máquina e consideramos as tarefas de forma gulosa, ordenadas por seus tempos iniciais. Para cada tarefa i, se houver uma máquina que puder tratar a tarefa i, então alocamos i para aquela máquina. Caso contrário, alocamos uma nova máquina, alocamos i para a mesma e repetimos o processo de escolha gulosa até ter considerado todas as tarefas de T.

O fato de que o algoritmo TaskSchedule funciona corretamente é definido pelo seguinte teorema:

Teorema 5.2: *dado um conjunto de n tarefas especificadas por seus tempos de início e fim, o algoritmo* TaskSchedule *encontra um escalonamento com um número mínimo de máquinas em tempo* $O(n \log n)$.

Prova: podemos mostrar que o algoritmo guloso simples acima, TaskSchedule, encontra um escalonamento ótimo com um número mínimo de máquinas.

Então, suponha que o algoritmo não funcione. Isto é, suponha que o algoritmo encontre um escalonamento sem conflitos usando k máquinas, mas que exista um escalonamento sem conflitos que usa $k - 1$ máquinas. Faça k ser a última máquina alocada pelo algoritmo e faça i ser a primeira tarefa alocada em k. Pela estrutura do algoritmo, quando escalonamos i, cada uma das máquinas de 1 até $k - 1$ contém tarefas conflitantes com i. Uma vez que elas conflitem, e pelo fato de trabalharmos com as tarefas ordenadas pelo tempo inicial, todas as tarefas que conflitam com i devem ter tempos iniciais menores ou iguais a s_i e tempos finais maiores do que s_i. Em outras palavras, essas tarefas não apenas conflitam com i, como conflitam entre si. Isso significa que temos k tarefas em nosso conjunto T que conflitam entre si, o que implica que é impossível escalonarmos todas as tarefas em T usando apenas $k - 1$ máquinas. Assim, k é o menor número de máquinas necessário para resolver o escalonamento de todas as tarefas de T.

Deixamos como exercício (R-5.2) a tarefa de mostrar como implementar o Algoritmo TaskSchedule em tempo $O(n \log n)$. ■

Consideramos várias outras aplicações do método guloso neste livro, incluindo dois problemas de compressão de *strings* (Seção 9.3) em que a abordagem gulosa leva a uma construção conhecida como codificação de Huffman, e algoritmos de grafos (Seção 7.3), em que a abordagem gulosa é usada para resolver os problemas de menor trajeto e menor conjunto de valores em uma árvore.

A próxima técnica que discutiremos é a de divisão e conquista, que é uma metodologia genérica para usar recursão no projeto de algoritmos eficientes.

5.2 Divisão e conquista

A técnica de ***divisão e conquista*** envolve a solução de um problema computacional particular dividindo-o em um ou mais subproblemas de menor tamanho, resolvendo cada subproblema recursivamente e, então, "juntando" ou "casando" as soluções dos subproblemas para produzir a solução do problema original.

Podemos modelar a abordagem de divisão e conquista usando um parâmetro n para indicar o tamanho do problema original e fazendo $S(n)$ denotar o problema em si. Solucionamos o problema $S(n)$ resolvendo a coleção de subproblemas $S(n_1)$, $S(n_2)$,..., $S(n_k)$, onde $n_i < n$ para $i = 1,..., k$ e, então, unindo a solução desses subproblemas.

Por exemplo, no algoritmo clássico *merge-sort* (Seção 4.1), $S(n)$ indica o problema de ordenar uma seqüência de n números. O *merge-sort* resolve o problema $S(n)$ dividindo-o em dois subproblemas $S(\lfloor n/2 \rfloor)$ e $S(\lceil n/2 \rceil)$, solucionando recursivamente esses problemas e, então, unindo os resultados em uma única seqüência ordenada que revela a solução de $S(n)$. Os passos de junção consomem tempo $O(n)$. Assim, o tempo total de execução do algoritmo *merge-sort* é $O(n \log n)$.

Como o caso do algoritmo *merge-sort*, a técnica genérica de divisão e conquista pode ser usada para construir algoritmos que executam mais rapidamente.

5.2.1 Equações recursivas de divisão e conquista

Para analisar o tempo de execução de um algoritmo de divisão e conquista, usamos equações recursivas (Seção 1.1.4). Assim, façamos uma função $T(n)$ indicar o tempo de execução do algoritmo a partir de uma entrada de tamanho n, e caracterizemos $T(n)$ usando uma equação que relacione $T(n)$ com os valores da função T para problemas menores do que n. No caso do algoritmo *merge-sort*, obtemos a equação recursiva

$$T(n) = \begin{cases} b & \text{se } n < 2 \\ 2T(n/2) + bn & \text{se } n \geq 2, \end{cases}$$

para uma constante $b > 0$, partindo da premissa simples de que n é uma potência de 2. Na verdade, no decorrer desta seção, usaremos a premissa de que n é uma potência apropriada, de maneira a evitar o uso de funções de arredondamento. Toda sentença assintótica que fizermos sobre equações recursivas será verdade, mesmo se relaxarmos esta premissa, mas justificar esse fato formalmente envolve provas longas e cansativas. Como observamos anteriormente, podemos mostrar que $T(n)$ é $O(n \log n)$ neste caso. Normalmente, entretanto, é possível deparar-nos com funções recursivas mais desafiadoras de resolver do que esta. Assim, é útil desenvolver algumas formas genéricas de resolver os tipos de equações recursivas que resultam da análise de algoritmos de divisão e conquista.

O método da substituição iterativa

Uma maneira de resolver uma equação recursiva de divisão e conquista é usar o método da ***substituição iterativa***, que é mais comumente conhecido como método "*plug-and-chug*". Usando este método, assumimos que um problema de tamanho n é muito grande e, então, substituímos a forma geral de recursão pelas ocorrências da função T no lado direito. Por exemplo, executar esta substituição com a equação de recursão do *merge-sort* resulta a equação

$$\begin{aligned} T(n) &= 2(2T(n/2^2) + b(n/2)) + bn \\ &= 2^2T(n/2^2) + 2bn. \end{aligned}$$

Usar a equação genérica para T novamente resulta a equação

$$\begin{aligned} T(n) &= 2^2(2T(n/2^3) + b(n/2^2)) + 2bn \\ &= 2^3T(n/2^3) + 3bn. \end{aligned}$$

O objetivo de se aplicar o método da substituição iterativa é esperar que, em algum momento, observemos um padrão que possa ser convertido em uma equação geral de forma fechada (com T aparecendo apenas no lado esquerdo). No caso da equação recursiva do *merge-sort*, a forma geral é

$$T(n) = 2^iT(n/2^i) + ibn.$$

Notemos que a forma geral dessa equação nos leva ao caso básico, $T(n) = b$, quando $n = 2^i$, isto é, quando $i = \log n$, o que implica que

$$T(n) = bn + bn \log n.$$

Em outras palavras, $T(n)$ é $O(n \log n)$. Em uma aplicação genérica da técnica de substituição iterativa, esperamos que seja possível determinar um padrão genérico para $T(n)$ e que possamos identificar quando a forma geral de $T(n)$ leva ao caso básico.

De um ponto de vista matemático, existe um momento no uso da técnica da substituição iterativa que envolve um "salto" lógico. Esse salto ocorre no momento em que tentamos caracterizar o padrão genérico emergindo de uma seqüência de substituições. Às vezes, como é o caso da equação do *merge-sort*, esse salto é simples. Outras vezes, entretanto, pode não ser tão óbvia a forma com a qual a equação se parece. Nestes casos, o salto pode ser mais complicado. Para estarmos totalmente seguros em fazer o salto, devemos justificar completamente a forma geral da equação, possivelmente usando indução. Combinado com tal justificativa, o método da substituição iterativa é completo e correto, e é normalmente uma forma útil para caracterizar equações recursivas. A propósito, o apelido "*plug-and-chug*", usado para descrever o método da substituição iterativa, vem da forma como compreende a conexão (*plug*) da parte recursiva de uma equação para $T(n)$ e então freqüentemente *chug** através de uma quantidade considerável de álgebra, de maneira a colocar essa equação em uma forma a partir da qual se possa inferir um padrão genérico.

A árvore de recursão

Outra maneira de caracterizar equações recursivas é usar o método da ***árvore de recursão***. Da mesma forma que o método da substituição interativa, essa técnica usa a substituição repetida para resolver uma equação recorrente, mas difere daquele método na medida em que, em vez de usar uma abordagem algébrica, é uma abordagem visual. Ao usar o método da árvore de recursão, desenhamos uma árvore R onde cada nodo representa uma substituição diferente na equação recorrente. Assim, cada nodo de R tem um valor do argumento n da função $T(n)$ associada. Além disso, associamos uma ***sobrecarga*** a cada nodo v de R, definida como o valor da parte não recursiva da equação recorrente de v. Para recorrências de divisão e conquista, a sobrecarga corresponde ao tempo de execução necessário para unir as soluções originárias dos filhos de v. A equação recorrente é então resolvida somando as sobrecargas associadas com todos os nodos de R. Isso é normalmente feito somando inicialmente os valores de cada nível de R, e então adicionando as somas parciais de cada um dos níveis.

Exemplo 5.3: *considere a seguinte equação recorrente:*

$$T(n) = \begin{cases} b & \text{se } n < 3 \\ 3T(n/3) + bn & \text{se } n \geq 3, \end{cases}$$

Essa é a equação recorrente que obtemos, por exemplo, pela modificação do algoritmo merge-sort de maneira a dividir a seqüência não ordenada em três seqüências de mesmo tamanho, ordenando recursivamente cada uma e, então, unindo as três seqüências ordenadas para produzir a versão ordenada da seqüência original. Na árvore de recursão R correspondente a esta recursão, cada nodo interno v possui três filhos, um tamanho e uma sobrecarga associados que correspondem ao tempo necessário para unir as soluções dos subproblemas produzidas pelos filhos de v. Apresentamos a árvore R na Figura 5.4. Observamos que as sobrecargas dos nodos de cada nível somam bn. Assim, notando que a profundidade de R é $\log_3 n$, *temos que* $T(n)$ *é* $O(n \log n)$.

* N. de T. Explode.

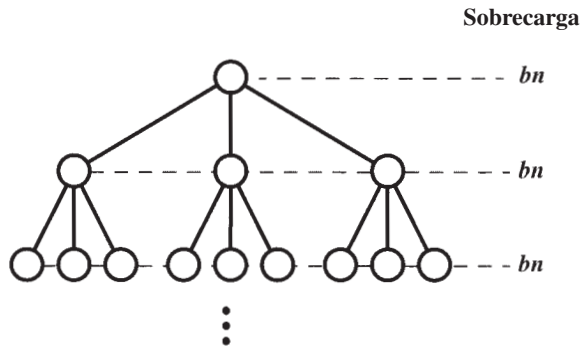


Figura 5.4 Árvore de recursão R usada no Exemplo 5.3, na qual mostramos a sobrecarga acumulada para cada nível.

O método adivinhe-e-teste

Outro método para resolver equações recursivas é a técnica ***adivinhe-e-teste***. Essa técnica envolve primeiramente fazer uma adivinhação razoável sobre a aparência da solução de forma fechada da equação recorrente e, então, justificar esse chute, geralmente por indução. Por exemplo, podemos usar o método do adivinhe-e-teste como uma espécie de "pesquisa binária" para encontrar bons limites superiores para uma equação recorrente. Se a justificativa do chute corrente falhar, então é possível que tenhamos de usar uma função de crescimento mais rápido, e se nosso chute se justifica com muita facilidade, então é possível que tenhamos de utilizar uma função de crescimento mais lento. Entretanto, o uso dessa técnica exige cuidado em cada passo matemático, quando se tenta justificar que uma certa hipótese está correta no que se refere a um dado "chute". Exploramos a aplicação desse método no exemplo que segue.

Exemplo 5.4: *considere a seguinte equação recorrente (assumindo o caso básico $T(n) = b$ para $n < 2$)*

$$T(n) = 2T(n/2) + bn \log n.$$

Essa é muito similar à equação recorrente da rotina de merge-sort; então, usaremos o seguinte como "chute" inicial:

$$\textit{Primeiro chute: } T(n) \leq cn \log n,$$

para alguma constante $c > 0$. Certamente podemos escolher c grande o suficiente para tornar isto verdadeiro para o caso básico; então, considere o caso quando $n \geq 2$. Se assumimos que nosso primeiro chute é uma hipótese indutiva que é verdadeira para tamanhos de entrada menores que n, então temos

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\leq 2(c(n/2)\log(n/2)) + bn \log n \\ &= cn(\log n - \log 2) + bn \log n \\ &= cn \log n - cn + bn \log n. \end{aligned}$$

Mas não existe maneira de tornar a última linha menor ou igual a $cn \log n$ para $n \geq 2$. Assim, esse primeiro "chute" não foi o suficiente. Vamos tentar então

$$\textit{Chute melhor: } T(n) \leq cn \log^2 n,$$

para uma constante $c > 0$. Novamente escolhemos c grande o suficiente para tornar isto verdadeiro para o caso base; então, consideremos o caso quando $n \geq 2$. Se assumimos que este chute é uma hipótese indutiva verdadeira para tamanhos de entrada menores do que n, então teremos

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\leq 2(c(n/2)\log^2(n/2)) + bn \log n \\ &= cn(\log^2 n - 2\log n + 1) + bn \log n \\ &= cn \log^2 n - 2cn \log n + bn \log n \\ &\leq cn \log^2 n \end{aligned}$$

desde que $c \geq b$. Assim, mostramos que $T(n)$ é de fato $O(n \log^2 n)$ neste caso.

Devemos ser cuidadosos ao usar esse método. Apenas pelo fato de que uma hipótese indutiva para $T(n)$ não funcionar, isso não implica necessariamente que outra, proporcional à primeira, não irá funcionar.

Exemplo 5.5: *consideremos a seguinte equação recorrente (assumindo o caso base $T(n) = b$ para $n < 2$):*

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

Essa recorrência é o tempo de execução para a construção bottom-up de um heap discutida na Seção 2.4.4, que mostramos ser $O(n)$. Entretanto, se tentarmos provar esse fato com a hipótese indutiva mais direta, teremos dificuldades. Considere o seguinte caso particular:

$$\textit{Primeiro chute: } T(n) \leq cn,$$

para uma constante $c > 0$. Podemos escolher c grande o suficiente para tornar isto verdadeiro para o caso base, certamente; então, consideremos o caso onde $n \geq 2$. Se assumimos este chute como uma hipótese indutiva que é verdadeira para tamanhos de entrada menores do que n, então teremos

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2(c(n/2)) + \log n \\ &= cn + \log n \end{aligned}$$

Mas não existe maneira de tornar a última linha menor ou igual a cn para $n \geq 2$. Assim, esse primeiro chute não foi suficiente, mesmo que $T(n)$ seja, na verdade, $O(n)$. Podemos mostrar ainda que este fato é verdadeiro usando

$$\textit{Chute melhor: } T(n) \leq c(n - \log n),$$

para uma constante $c > 0$. Podemos, novamente, escolher c grande o suficiente para tornar isto verdadeiro para o caso base; na realidade, podemos mostrar que é verdadeiro para qualquer $n < 8$. Então consideremos o caso onde $n \geq 8$. Se assumimos este chute como uma hipótese indutiva que é verdadeira para os casos onde o tamanho de entrada é menor do que n, então temos

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

dado que $c \geq 3$ *e* $n \geq 8$. *Assim, mostramos que* $T(n)$ *é, na verdade,* $O(n)$ *neste caso.*

O método adivinhe-e-teste pode ser usado para estabelecer tanto um limite superior como inferior para a complexidade assintótica de uma equação recursiva. Mesmo assim, como os exemplos demonstraram, requer que tenhamos desenvolvido certa experiência com o uso de indução matemática.

O método mestre

Cada um dos métodos descritos anteriormente para resolver equações recorrentes é *ad hoc* e requer uma certa sofisticação matemática para poder ser usado de forma efetiva. Existe, entretanto, um método para resolver equações de divisão e conquista recorrentes que é mais geral e não exige o uso explícito de indução matemática para ser aplicado corretamente – o ***método mestre***. O método mestre é um "guia culinário" para determinar a caracterização assintótica de uma grande variedade de equações recorrentes. Na verdade, é usado com equações recorrentes da forma

$$T(n) = \begin{cases} c & \text{se } n < d \\ aT(n/b) + f(n) & \text{se } n \geq d, \end{cases}$$

onde $d \geq 1$ é uma constante inteira; $a > 0$, $c > 0$ e $b > 1$ são constantes reais; e $f(n)$é uma função positiva para $n \geq d$. Essa equação recorrente se encaixa na análise do algoritmo de divisão e conquista que divide um dado problema em a subproblemas cujos tamanhos são no máximo n/b, resolve cada um recursivamente e então "junta" a solução dos subproblemas em uma única. A função $f(n)$ nesta equação mostra o tempo adicional necessário para dividir o problema em subproblemas e juntar suas soluções em uma única. Cada uma das equações recorrentes acima usa essa forma, mostrada anteriormente neste livro, para analisar o algoritmo de divisão e conquista. Assim, é na verdade uma forma genérica de equação recorrente de divisão e conquista.

O método mestre para resolver essa equação recorrente envolve simplesmente escrever a resposta com base em um dos três casos que se aplicam. Distingue-se cada um desses casos pela comparação de $f(n)$ com uma função especial $n^{\log_b a}$ (mostraremos depois por que essa função especial é tão importante).

Teorema 5.6 [O Teorema Mestre]: *faça* $f(n)$ *e* $T(n)$ *serem definidos como anteriormente.*

1. *Se existe uma constante pequena* $\varepsilon > 0$, *tal que* $f(n)$ é $O(n^{\log_b a - \varepsilon})$, *então* $T(n)$ *é* $\Theta(n^{\log_b a})$.
2. *Se existe uma constante* $k \geq 0$, *tal que* $f(n)$ *é* $\Theta(n^{\log_b a}\log^k n)$, *então* $T(n)$ *é* $\Theta(n^{\log_b a}\log^{k+1} n)$.
3. *Se existem constantes pequenas* $\varepsilon > 0$ e $\delta < 1$, tal que $f(n)$ é $\Omega(n^{\log_b a + \varepsilon})$ *e* $af(n/b) \leq \delta f(n)$ *para* $n \geq d$, $T(n)$ *é* $\Theta(f(n))$.

O Caso 1 caracteriza a situação na qual $f(n)$ é polinomialmente menor do que a função especial $n^{\log_b a}$. O Caso 2 caracteriza a situação na qual $f(n)$ é assintoticamente próxima à função especial. O caso 3 caracteriza a situação na qual $f(n)$ é polinomialmente maior do que a função especial.

Ilustramos o uso do método mestre através de alguns exemplos (em cada um, assumimos que $T(n) = c$ para $n < d$, para constantes $c \geq 1$ e $d \geq 1$).

Exemplo 5.7: *consideremos a recorrência*

$$T(n) = 4T(n/2) + n.$$

Neste caso, $n^{\log_b a} = n^{\log_2 4} = n^2$. *Assim, estamos no Caso 1, pois* $f(n)$ *é* $O(n^{2-\varepsilon})$ *para* $\varepsilon = 1$. *Isso significa que, pelo método mestre,* $T(n)$ *é* $\Theta(n^2)$.

Exemplo 5.8: *consideremos a recorrência*

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

que é uma das recorrências vistas anteriormente. Neste caso, $n^{\log_b a} = n^{\log_2 2} = n$. *Assim, estamos no Caso 2, com* $k = 1$ *pois* $f(n)$ *é* $\Theta(n \log n)$. *Isso significa que, pelo método mestre,* $T(n)$ *é* $\Theta(n \log^2 n)$.

Exemplo 5.9: *consideremos a recorrência*

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

que é a recorrência para uma soma geometricamente decrescente que se inicia em n. Neste caso, $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. *Assim, estamos no Caso 3, pois* $f(n)$ *é* $\Theta(n \log n)$ *para* $\varepsilon = 1$, *e* $af(n/b) + n/3 = (1/3)f(n)$. *Isso significa que, pelo método mestre,* $T(n)$ é $\Theta(n)$.

Exemplo 5.10: *consideremos a recorrência*

$$T(n) = 9T(n/3) + n^{2,5},$$

Neste caso, $n^{\log_b a} = n^{\log_3 9} = n^2$. *Assim, estamos no Caso 3, uma vez que* $f(n)$ *é* $\Omega(n^{2+\varepsilon})$ (*para* $\varepsilon = 1/2$) *e* $af(n/b) + 9(n/3)^{2,5} = (1/3)f(n)$. *Isso significa que, pelo método mestre,* $T(n)$ *é* $\Theta(n^{2,5})$.

Exemplo 5.11: *por fim, consideremos a recorrência*

$$T(n) = 2T(n^{1/2}) + \log n.$$

Infelizmente, essa equação não se encontra em um formato que nos permita usar o método mestre. Podemos, entretanto, colocá-la de tal forma introduzindo a variável $k = \log n$, *o que nos permite escrever*

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

Substituindo nesta equação $S(k) = T(2^k)$, *temos que*

$$S(k) = 2S(k/2) + k.$$

Agora, essa equação de recorrência nos permite usar o método mestre que especifica que $S(k)$ *é* $O(k \log k)$. *Restaurar* $T(n)$ *implica que* $T(n)$ *é* $O(\log n \log \log n)$.

Em vez de provar com rigor o Teorema 5.6, discutiremos as justificativas por trás do método mestre em um nível mais alto.

Se aplicarmos o método da substituição iterativa à equação genérica de divisão e conquista, obtemos

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2) + af(n/b) + f(n) = a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&\quad \vdots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i),
\end{aligned}
$$

onde a última substituição é baseada na identidade $a^{\log_b n} = n^{\log_b a}$. Na verdade, essa equação é a origem da função especial. Dada essa caracterização de forma fechada para $T(n)$, podemos intuitivamente ver como cada um dos três casos é derivado. O Caso 1 se origina em uma situação em que $f(n)$ é pequeno, e o primeiro termo acima domina. O Caso 2 indica a situação em que cada um dos termos do somatório acima é proporcional aos demais; logo, a caracterização de $T(n)$ é $f(n)$ vezes um fator logarítmico. Finalmente, o Caso 3 indica a situação em que o primeiro termo é menor do que o segundo, e o somatório acima é uma soma de termos geometricamente decrescentes que inicia em $f(n)$; assim, $T(n)$ é proporcional a $f(n)$.

A prova do Teorema 5.6 formaliza essa intuição, mas em vez de fornecer detalhes desta prova, apresentaremos, em seguida, duas aplicações do método mestre.

5.2.2 Multiplicação de inteiros

Nesta subseção, consideraremos o problema da multiplicação de ***inteiros grandes***, isto é, inteiros representados por uma grande quantidade de *bits* que não podem ser manipulados diretamente pela unidade aritmética de um processador. A multiplicação de inteiros grandes tem aplicações em segurança de dados, onde números desse tipo são usados em esquemas de criptografia.

Dados dois inteiros grandes I e J representados por n *bits* cada, podemos calcular facilmente $I + J$ e $I - J$ em tempo $O(n)$. Calcular o produto $I \cdot J$ usando o algoritmo aprendido na escola requer, entretanto, tempo $O(n^2)$. No restante desta seção, mostraremos que, usando a técnica de divisão e conquista, podemos projetar um algoritmo de tempo subquadrático para multiplicar dois números inteiros de n *bits*.

Vamos assumir que n é uma potência de 2 (se este não for o caso, podemos completar I e J com zeros). Podemos, por esta razão, dividir as representações de *bits* de I e J pela metade, com uma metade representando os *bits* de ***mais alta ordem*** e a outra representando os *bits* de ***mais baixa ordem***. Em especial, se dividimos I em I_h e I_l, e J em J_h e J_l, então

$$
\begin{aligned}
I &= I_h 2^{n/2} + I_l, \\
J &= J_h 2^{n/2} + J_l.
\end{aligned}
$$

Observamos também que multiplicar um número binário I por uma potência de 2, 2^k, é trivial – simplesmente envolve o deslocamento para a esquerda (isto é, na direção dos *bits* de mais alta ordem) do número I de k posições de *bits*. Assim, uma vez que uma operação de deslocamento para a esquerda consome tempo constante, multiplicar um inteiro por 2^k consome tempo $O(k)$.

Vamos nos concentrar no problema do cálculo do produto $I \cdot J$. Dada a expansão de I e J acima, podemos reescrever I e J como

$$I \cdot J = (I_h 2^{n/2} + I_l) \cdot (J_h 2^{n/2} + J_l) = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l.$$

Assim, podemos calcular $I \cdot J$ aplicando um algoritmo de divisão e conquista que divide a representação de *bits* de I e J pela metade, calcula recursivamente o produto dos produtos de cada $n/2$ *bits* (como descrito acima) e, então, junta as soluções desses subprodutos em tempo $O(n)$ usando adições e multiplicações de potências de 2. Podemos encerrar a recursão quando obtivermos a multiplicação de dois números de 1 *bit*, o que é trivial. Esse algoritmo de divisão e conquista tem um tempo de execução que pode ser caracterizado pela seguinte recorrência (para $n \geq 2$):

$$T(n) = 4T(n/2) + cn,$$

para alguma constante $c > 0$. Podemos então aplicar o teorema mestre para observar a função especial $n^{\log_b a} = n^{\log_2 4} = n^2$ neste caso; assim, estamos no Caso 1, e $T(n)$ é $\Theta(n^2)$. Infelizmente, isso não é melhor do que o algoritmo aprendido na escola.

O método mestre nos fornece algumas pistas sobre como podemos melhorar esse algoritmo. Se pudermos reduzir o número de chamadas recursivas, então reduziremos a complexidade da função especial usada no teorema mestre, que é, na verdade, o fator dominante de nosso tempo de execução. Felizmente, sendo um pouco mais espertos na definição dos subproblemas a serem resolvidos recursivamente, poderemos, de fato, reduzir o número de chamadas recursivas em um. Em especial, consideremos o produto

$$(I_h - I_l) \cdot (J_l - J_h) = I_h J_l - I_l J_l - I_h J_h + I_l J_h.$$

Esse é, na verdade, um produto estranho para se considerar, mas tem uma propriedade interessante. Quando expandido, contém dois produtos que queremos calcular (a saber, $I_h J_l$ e $I_l J_h$) e dois produtos que podemos calcular recursivamente (a saber, $I_h J_h$ e $I_l J_l$). Assim, podemos calcular $I \cdot J$ como segue:

$$I \cdot J = I_h J_h 2^n + [(I_h - I_l) \,.\, (J_l - J_h) + I_h J_h + I_l J_h] 2^{n/2} + I_l J_l.$$

Esse cálculo requer a computação recursiva de três produtos de $n/2$ *bits* cada, mais $O(n)$ trabalho adicional, resultando, assim, em um algoritmo de divisão e conquista com um tempo de execução caracterizado pela seguinte equação de recorrência (para $n \geq 2$):

$$T(n) = 3T(n/2) + cn$$

para uma constante $c > 0$.

Teorema 5.12: *podemos multiplicar dois inteiros de n bits em tempo* $O(n^{1,585})$.

Prova: aplicamos o teorema mestre usando a função especial $n^{\log_b a} = n^{\log_2 3}$; assim estamos no Caso 1, e $T(n)$ é $\Theta(n^{\log_2 3})$, que é $O(n^{1,585})$. ■

Usando divisão e conquista, projetamos um algoritmo para multiplicação de inteiros que é assintoticamente mais rápido do que o método direto de tempo quadrático. Na verdade, podemos fazer melhor do que isso obtendo um tempo de execução que é "quase" $O(n \log n)$, usando um algoritmo de divisão e conquista mais complexo chamado ***transformada rápida de Fourier***, que será visto na Seção 10.4.

5.2.3 Multiplicação de matrizes

Suponhamos que sejam dadas duas matrizes X e Y de tamanho $n \times n$, e que queremos calcular seu produto $Z = XY$, que é definido como

$$Z[i,j] = \sum_{k=0}^{n-1} X[i,k] \cdot Y[k,j],$$

cuja equação nos leva a um algoritmo simples que consome tempo $O(n^3)$.

Outra forma de enxergar esse produto é em função de submatrizes. Isto é, vamos assumir que n é potência de 2 e vamos partir X, Y e Z em quatro matrizes $(n/2) \times (n/2)$, de forma que podemos reescrever $Z = XY$ como

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Assim,

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH. \end{aligned}$$

Podemos usar esse conjunto de equações em um algoritmo de divisão e conquista que calcula $Z = XY$ computando I, J, K e L a partir de seus subarranjos A até G. Pelas equações acima, podemos calcular I, J, K e L a partir de oito produtos de matrizes recursivos calculados sobre subarranjos $(n/2) \times (n/2)$, mais quatro adições que podem ser feitas em tempo $O(n^2)$. Assim, o conjunto de equações acima nos leva a um algoritmo de divisão e conquista cujo tempo de execução $T(n)$ é caracterizado pela recorrência

$$T(n) = 8T(n/2) + bn^2,$$

para uma constante $b > 0$. Infelizmente, essa equação implica que $T(n)$ é $O(n^3)$ pelo teorema mestre; conseqüentemente, esse algoritmo não é melhor do que o algoritmo direto de multiplicação de matrizes.

Por outro lado, existe um algoritmo conhecido como **Algoritmo de Strassen**, que organiza a aritmética envolvida nos subarranjos A até G de maneira que podemos calcular I, J, K e L usando apenas sete multiplicações recursivas de matrizes. É um tanto quanto misteriosa a forma pela qual Strassen descobriu essas equações, mas podemos verificar com facilidade que funcionam corretamente.

Começamos o Algoritmo de Strassen definindo os sete produtos das submatrizes:

$$\begin{aligned} S_1 &= A(F - H) \\ S_2 &= (A + B)H \\ S_3 &= (C + D)E \\ S_4 &= D(G - E) \\ S_5 &= (A + D)(E + H) \\ S_6 &= (B - D)(G + H) \\ S_7 &= (A - C)(E + F) \end{aligned}$$

Dados esses sete produtos de submatrizes, podemos calcular I como

$$\begin{aligned} I &= S_5 + S_6 + S_4 - S_2 \\ &= (A+D)(E+H) + (B-D)(G+H) + D(G-E) - (A+B)H \\ &= AE + DE + AH + DH + BG - DG + BH - DH + DG - DE - AH - BH \\ &= AE + BG \end{aligned}$$

Podemos calcular J como

$$\begin{aligned} J &= S_1 + S_2 \\ &= A(F-H) + (A+B)H \\ &= AF - AH + AH + BH \\ &= AF + BH \end{aligned}$$

Podemos calcular K como

$$\begin{aligned} K &= S_3 + S_4 \\ &= (C+D)E + D(G-E) \\ &= CE + DE + DG - DE \\ &= CE + DG \end{aligned}$$

Por fim, podemos calcular L como

$$\begin{aligned} L &= S_1 - S_7 - S_3 + S_5 \\ &= A(F-H) - (A-C)(E+F) - (C+D)E + (A+D)(E+H) \\ &= AF - AH - AE + CE - AF + CF - CE - DE + AE + DE + AH + DH \\ &= CF + DH \end{aligned}$$

Assim, podemos calcular $Z = XY$ usando sete multiplicações recursivas de matrizes de tamanho $(n/2) \times (n/2)$. Logo, podemos caracterizar o tempo de execução $T(n)$ como

$$T(n) = 7T(n/2) + bn^2,$$

para uma constante $b > 0$. Assim, pelo teorema mestre, temos o que segue:

Teorema 5.13: *podemos multiplicar duas matrizes $n \times n$ em tempo $O(n^{\log 7})$.*

Desta forma, com um pouco de complicação adicional, podemos executar a multiplicação de matrizes $n \times n$ em tempo $O(n^{2,808})$, o que corresponde a $o(n^3)$. Tão complicados como o Algoritmo de Strassen admite ser, existem, na verdade, algorimos de multiplicação de matrizes muito mais complicados, com tempos de execução mais baixos, tais como $O(n^{2,376})$.

5.3 Programação dinâmica

Nesta seção, discutiremos a técnica de projeto de algoritmos conhecida como ***programação dinâmica***. Essa técnica é similar à técnica de divisão e conquista na medida em que pode ser aplicada a uma grande gama de problemas. Conceitualmente, entretanto, as

duas são diferentes, porque a técnica de divisão e conquista pode ser facilmente explicada em uma frase ou duas e exemplificada. A programação dinâmica requer um pouco mais de explicação e vários exemplos antes de poder ser completamente entendida.

Acreditamos, entretanto, que o esforço extra para poder entender a programação dinâmica vale o investimento. Existem poucas técnicas de algortimos que podem tratar de um problema cuja solução aparente requer tempo exponencial e produzir algoritmos de tempo polinomial para resolvê-los. A programação dinâmica é uma dessas técnicas. Além disso, os algoritmos resultantes da aplicação da técnica de programação dinâmica são, em geral, mais simples – geralmente necessitam de poucas linhas de código para descrever alguns laços aninhados para preencher uma tabela.

5.3.1 Produto de matrizes encadeadas

Em vez de iniciar com uma explicação geral sobre a técnica de progamação dinâmica, começamos apresentando um exemplo clássico e concreto. Suponha uma coleção de n matrizes das quais desejamos calcular o produto:

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1},$$

onde A_i é uma matriz $d_i \times d_{i+1}$ para $i = 0, 1, 2,..., n - 1$. Usando o algoritmo de multiplicação de matrizes padrão (que é o que iremos usar) para multiplicar uma matriz B, $d \times e$, por uma matriz F, $e \times f$, calculamos a entrada (i, j) do produto por

$$\sum_{k=0}^{e-1} B[i,k] \cdot C[k,j].$$

Essa definição implica que a multiplicação de matrizes é associativa, ou seja, que $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Assim, podemos colocar parênteses na expressão de qualquer maneira que obteremos sempre o mesmo resultado. Entretanto, não iremos executar o mesmo número de multiplicações primitivas (isto é, escalares) em cada parentetização, como pode ser visto no exemplo a seguir.

Exemplo 5.14: *seja B uma matriz 2×10, seja C uma matriz 10×50 e seja D uma matriz 50×20. Calcular $B \cdot (C \cdot D)$ requer $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10400$ multiplicações, enquanto calcular $(B \cdot C) \cdot D$ requer $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ multiplicações.*

O problema do ***produto das cadeias de matrizes*** é determinar a parentetização da expressão que define o produto A que minimize o número total de multiplicações escalares a serem executadas. É claro que uma forma de resolver esse problema é simplesmente enumerar todas as formas possíveis de colocar parênteses na expressão que define o produto A e determinar o número de multiplicações necessárias para cada uma. Infelizmente, o conjunto de todas as diferentes possibilidades de parentetização para uma expressão A é igual em número ao conjunto de árvores binárias que têm n nodos externos. Esse número é exponencial proporcionalmente a n. Assim, esse algoritmo direto ("força bruta") executa em tempo exponencial, uma vez que existe um número exponencial de formas de parentetizar uma expressão aritmética associativa (o número é igual ao n-ésimo ***número de Catalan***, que corresponde a $\Omega(4^n/n^{3/2})$).

Definição de subproblemas

Podemos, entretanto, melhorar significativamente a performance obtida pelo algoritmo de força bruta fazendo algumas observações sobre a natureza do problema do produto das cadeias de matrizes. A primeira observação é que o problema pode ser dividido em ***subproblemas***. Neste caso, podemos definir diferentes subproblemas, cada um dos quais visando ao cálculo da melhor parentetização da subexpressão $A_i \cdot A_{i+1} \cdots A_j$. Como notação resumida, usamos $N_{i,j}$ para indicar o número mínimo de multiplicações necessárias para calcular essa subexpressão. Assim, o produto de cadeia de matrizes original pode ser caracrerizado como o cálculo do valor de $N_{0,n-1}$. Essa observação é importante, mas necessitamos mais para poder aplicar a técnica de programação dinâmica.

Caracterização de soluções ótimas

A outra observação importante que podemos fazer a respeito do problema da multiplicação da cadeia de matrizes é a possibilidade de caracterizar uma solução ótima para um subproblema particular em termos das soluções ótimas de seus subproblemas. Chamamos essa propriedade de condição dos ***subproblemas ótimos***.

No caso do produto da cadeia de matrizes, observamos que, não importando como colocamos parênteses numa subexpressão, é preciso que exista uma multiplicação que seja a final. Isto é, a parentetização completa da subexpressão $A_i \cdot A_{i+1} \cdots A_j$ tem de ser da forma $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, para um $k \in \{i, i+1, \ldots, j-1\}$. Além disso, independentemente de qual seja o valor correto para k, os produtos $(A_i \cdots A_k)$ e $(A_{k+1} \cdots A_j)$ também devem ser resolvidos de maneira ótima. Se não fosse assim, então existiria uma solução ótima global que teria um desses subproblemas resolvidos de maneira quase ótima. Mas isso é impossível, uma vez que, neste caso, poderíamos reduzir o número total de multiplicações através da substituição da solução corrente do subproblema por uma solução ótima do subproblema. Essa observação tem como conseqüência uma forma de definir explicitamente o problema da otimização para $N_{i,j}$ em função de outras soluções ótimas para os subproblemas. Na verdade, podemos calcular $N_{i,j}$ considerando cada lugar k onde podemos colocar a multiplicação final e obter o valor mínimo entre todos os possíveis.

Projeto de um algoritmo de programação dinâmica

A discussão anterior implica podermos caracterizar a solução ótima para um subproblema $N_{i,j}$ como

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\},$$

onde percebemos que

$$N_{i,j} = 0,$$

uma vez que nenhum esforço é necessário para uma subexpressão que envolva apenas uma matriz. Assim, $N_{i,j}$ é o mínimo, consideradas todas as formas de executar a multiplicação final, do número de multiplicações necessárias para calcular cada subexpressão, mais o número de multiplicações necessárias para executar a multiplicação final das matrizes.

A equação para $N_{i,j}$ é parecida com as equações recorrentes que derivamos para os algoritmos de divisão e conquista, mas é uma semelhança apenas superficial, pois existe um aspecto da equação $N_{i,j}$ que dificulta usar divisão e conquista para calcular $N_{i,j}$. Em particular, existem ***subproblemas compartilhados*** que nos impedem de dividir o problema em subproblemas completamente independentes (como seria necessário para aplicar divisão e conquista). Podemos, entretanto, usar a equação de $N_{i,j}$ para derivar um algoritmo eficiente calculando $N_{i,j}$ valores em uma abordagem *bottom-up* e armazenando as soluções intermediárias em uma tabela de $N_{i,j}$ valores. Podemos iniciar simplesmente atribuindo $N_{i,j} = 0$, para $i = 0, 1,..., n - 1$. Aplicamos, então, a equação geral de $N_{i,j}$ para calcular $N_{i,i+1}$ valores, uma vez que dependem apenas de $N_{i,i}$ e $N_{i+1,i+1}$ valores que estão disponíveis. Dados os $N_{i,i+1}$ valores, podemos então calcular os valores $N_{i,i+2}$ e assim por diante. Assim, podemos construir $N_{i,j}$ valores a partir dos valores previamente calculados até finalmente calcular o valor $N_{0,n-1}$, que é o número que estamos procurando. Os detalhes dessa solução usando ***programação dinâmica*** são fornecidos no Algoritmo 5.5.

Algoritmo MatrixChain($d_0,..., d_n$):
  Entrada: seqüência $d_0,..., d_n$ de inteiros
  Saída: para $i,j = 0,..., n - 1$, o número mínimo de multiplicações $N_{i,j}$ necessárias
    para calcular o produto $A_i \cdot A_{i+1} \cdots A_j$, onde A_k é uma matriz $d_k \times d_{k+1}$
  para $i \leftarrow 0$ até $n - 1$ **faça**
    $N_{i,j} \leftarrow 0$
  para $b \leftarrow 1$ até $n - 1$ **faça**
    para $i \leftarrow 0$ até $n - b - 1$ **faça**
      $j \leftarrow i + b$
      $N_{i,j} \leftarrow +\infty$
      para $k \leftarrow i$ até $j - 1$ **faça**
        $N_{i,j} \leftarrow \min\{ N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$.

Algoritmo 5.5 Algoritmo de programação dinâmica para o problema do produto de matrizes encadeadas.

Análise do algoritmo do produto de matrizes encadeadas

Logo, podemos calcular $N_{0,n-1}$ com um algoritmo que consiste basicamente em três laços aninhados: o laço mais externo é executado n vezes; o laço do meio é executado no máximo n vezes; e o laço mais interno é também executado no máximo n vezes. Conseqüentemente, o tempo total de execução desse algoritmo é $O(n^3)$.

Teorema 5.15: *dado um produto de matrizes encadeadas de duas dimensões, podemos calcular a parentetização desta cadeia que obtém número mínimo de multiplicações escalares em tempo* $O(n^3)$.

Prova: demostramos acima como podemos calcular o ***número*** ótimo de multiplicações escalares. Mas como iremos recuperar a verdadeira parentetização?

O método para calcular a parentetização é, na verdade, bastante direto. Modificamos o algoritmo para calcular $N_{i,j}$ valores de forma que a todo momento que encontramos um novo valor mínimo para $N_{i,j}$, armazenamos, com $N_{i,j}$, o índice k que nos permite obter este mínimo. ■

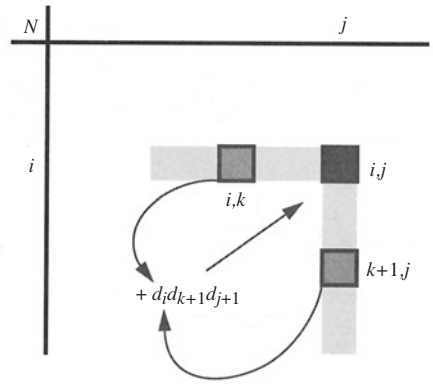


Figura 5.6 Ilustração da forma como o algoritmo de programação dinâmica para o problema do produto de matrizes encadeadas preenche o arranjo *N*.

Na Figura 5.6, ilustramos a forma como a solução de programação dinâmica para o problema do produto de matrizes encadeadas preenche o arranjo *N*.

Agora que trabalhamos um exemplo completo de uso do método de programação dinâmica, discutiremos os aspectos gerais da técnica de programação dinâmica em sua aplicação a outros problemas.

5.3.2 A técnica genérica

A técnica de programação dinâmica é usada sobretudo em problemas de ***otimização***, em que queremos encontrar a "melhor maneira" de fazer alguma coisa. Freqüentemente, a quantidade de formas diferentes para se fazer "alguma coisa" é exponencial, de maneira que usar força bruta para encontrar a melhor é computacionalmente impraticável, exceto para os problemas pequenos. Podemos aplicar a técnica de programação dinâmica nessas situações, entretanto, se o problema tiver uma certa estrutura que pode ser explorada. Essa estrutura envolve os três componentes a seguir:

Subproblemas simples: é necessário que exista uma forma de quebrar o problema de otimização geral em subproblemas, cada um com estrutura semelhante à do problema original. Além disso, deve existir uma forma simples de definir subproblemas usando apenas alguns índices, tais como *i*, *j*, *k*, e assim por diante.

Solução ótima dos subproblemas: uma solução ótima para o problema global deve ser uma composição das soluções ótimas para os subproblemas usando-se operações combinatórias relativamente simples. Não seremos capazes de encontrar uma solução global ótima que contenha subproblemas que não atingem o mesmo nível.

Sobreposição de subproblemas: soluções ótimas para problemas não relacionados podem conter subproblemas em comum. Na verdade, tal sobreposição melhora a

eficiência do algoritmo de programação dinâmica que armazena as soluções dos subproblemas.

Agora que apresentamos os componentes de um algoritmo de programação dinâmica, apresentaremos outro exemplo de uso.

5.3.3 O problema da mochila 0-1

Suponha que uma amante de caminhadas está para percorrer uma trilha através de uma floresta tropical carregando uma única mochila. Suponha que ela saiba o peso máximo W que pode carregar e que tenha um conjunto S de n coisas úteis que poderia potencialmente levar consigo, tais como uma cadeira dobrável, uma barraca e um exemplar deste livro. Vamos supor que cada item i tenha um peso w_i e um valor correspondente a seu benefício b_i que é um valor que a praticante de caminhadas atribui a um dado item i. Seu problema, é claro, é otimizar o valor total do conjunto T de itens que carrega consigo sem extrapolar o limite de peso W. Isto é, ela tem o seguinte objetivo:

$$\text{Maximizar } \sum_{i \in T} b_i \qquad \text{sujeito a } \sum_{i \in T} w_i \leq W.$$

Seu problema é uma instância do ***problema da mochila* 0-1**, chamado de "0-1" porque cada um dos itens só pode ser aceito ou rejeitado. Consideramos uma versão fracionária deste problema na Seção 5.1.1 e estudamos como os problemas de mochila surgem no contexto de leilões na Internet no Exercício R-5.12.

Primeira tentativa de caracterizar os subproblemas

Podemos resolver o problema da mochila 0-1 em tempo $\Theta(2^n)$, é claro, através da enumeração de todos os subconjuntos de S e da seleção daquele que tem o maior benefício total entre todos cujo peso total não exceda W. Esse seria, entretanto, um algoritmo ineficiente. Felizmente, podemos derivar um algoritmo de programação dinâmica para o problema da mochila 0-1 que executa mais rapidamente na maioria dos casos.

Como muitos problemas de programação dinâmica, uma das partes mais difíceis em projetar o algoritmo para o problema da mochila 0-1 é encontrar uma boa caracterização para os subproblemas (de forma que satisfaçamos às três propriedades de um algoritmo de programação dinâmica). Para simplificar a discussão, numeramos os itens de S como $1, 2, \ldots, n$ e vamos definir para cada $k \in \{1, 2, \ldots, n\}$ o subconjunto

$$S_k = \{\text{itens de } S \text{ rotulados } 1, 2, \ldots, k\}$$

Uma possibilidade é definirmos os subproblemas usando um parâmetro k de forma que o subproblema k seja a melhor maneira de preencher a mochila usando apenas os itens do conjunto S_k. Essa é uma definição válida de subproblema, mas não está claro como definir uma solução ótima para o índice k em função de soluções ótimas de subproblemas. Nossa esperança é que possamos derivar uma equação que nos forneça a solução usando itens de S_{k-1} e consideremos como adicionar o item k à mesma. Infelizmente, se teimarmos nessa definição para os subproblemas, então esta abordagem estará fadada ao fracasso. Assim, como mostramos na Figura 5.7, se usarmos essa caracterização para os subproblemas, então uma solução ótima para o problema global pode, na verdade, conter um subproblema que não atinge este nível.

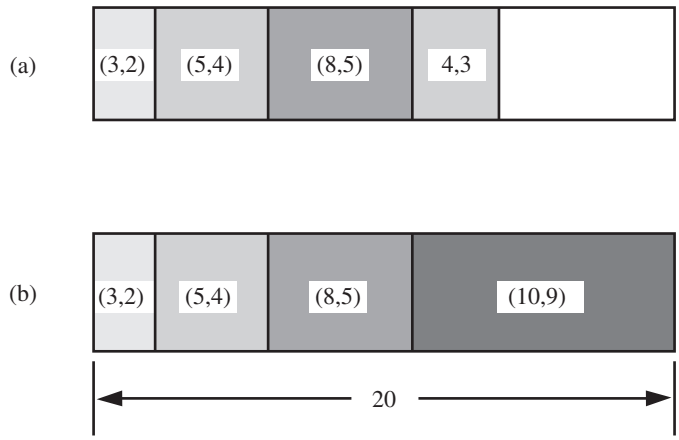


Figura 5.7 Exemplo que mostra como nossa primeira abordagem para definir um subproblema do problema da mochila não funciona. O conjunto S consiste em cinco itens denotados pelos pares (***peso*, *benefício***), a saber (3,2), (5,4), (8,5), (4,3) e (10,9). O peso total máximo é W = 20: (a) melhor solução com os primeiros quatro itens; (b) melhor solução com os primeiros cinco itens. Sombreamos cada item proporcionalmente a seu benefício.

Uma melhor caracterização do subproblema

Uma das razões pelas quais a definição dos subproblemas apenas em função de um índice k está fadada ao fracasso é que não existe informação suficiente representada em um subproblema para prover alguma ajuda na solução do problema global de otimização. Podemos corrigir essa dificuldade, entretanto, acrescentando um segundo parâmetro w. Vamos, então, formular cada subproblema como sendo o cálculo de $B[k,w]$, que é definido como o valor total máximo de um subconjunto S_k entre todos os subconjuntos que têm seus pesos totais ***até*** w. Teremos $B[0,w] = 0$ para cada $w \leq W$ e derivaremos a seguinte relação para o caso geral:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{se } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{em caso contrário.} \end{cases}$$

Ou seja, o melhor conjunto S_k cujo peso total w é tanto o melhor subconjunto de S_{k-1} que tem o peso total w quanto o melhor subconjunto de S_{k-1} que tem o peso total $w - w_k$ mais o item k. Uma vez que o melhor subconjunto de S_k com o peso total w deva ou não conter o item k, uma destas duas escolhas será a melhor. Assim, temos uma definição de subproblema que é simples (envolve apenas dois parâmetros, k e w) e satisfaz à condição de otimização do subproblema. Além disso, existe uma sobreposição de subproblemas para a forma ótima de somar até w ao peso que pode ser usado em muitos subproblemas futuros.

Na derivação de um algoritmo para essa definição, podemos fazer uma observação adicional, a saber, que a definição de $B[k,w]$ é construída a partir de $B[k-1,w]$ e possivelmente $B[k-1,w-w_k]$. Assim, podemos implementar esse algoritmo usando apenas um único arranjo B, que atualizamos em cada uma das séries de iterações indexadas por um parâmetro k de maneira que, ao final de cada iteração, $B[w] = B[w,k]$. Isso nos leva ao Algoritmo 5.8 (01Knapsack).

Algoritmo 01Knapsack(S,W):
  Entrada: conjunto S de n itens, tais que o item i tem um benefício positivo b_i e um peso inteiro positivo w_i; um inteiro positivo que corresponde ao peso total W
  Saída: para $w = 0,..., W$, o benefício máximo $B[w]$ de um subconjunto de S com peso total w.
  para $w \leftarrow 0$ até W **faça**
    $B[w] \leftarrow 0$
  para $k \leftarrow 1$ até n **faça**
    para $w \leftarrow W$ decrescendo até w_k **faça**
      se $B[w - w_k] + b_k > B[w]$ **então**
        $B[w] \leftarrow B[w - w_k] + b_k$

Algoritmo 5.8 Algoritmo de programação dinâmica para resolver o problema da mochila 0-1.

Análise do algoritmo de programação dinâmica para o problema da mochila 0-1

O tempo de execução do algoritmo 01Knapsack é determinado pelos dois laços aninhados, onde o mais externo itera n vezes e o mais interno itera no máximo W vezes. Após os laços estarem completos, podemos determinar o valor ótimo localizando o valor $B[w]$ que é maior entre todos os $w \leq W$. Assim, temos o seguinte:

Teorema 5.16: *dado um inteiro W e um conjunto S de n itens, cada qual tendo um benefício positivo e um peso inteiro positivo, podemos encontrar o subconjunto de S de maior benefício com peso total de no máximo W em tempo $O(nW)$.*

Prova: partimos do Algoritmo 5.8 (01Knapsack) para construir o ***valor*** do subconjunto de benefício máximo de S que tem peso total de no máximo W, usando um arranjo B de valores de benefício. Podemos facilmente converter nosso algoritmo, entretanto, em um que converta os itens em um subconjunto melhor. Deixamos os detalhes dessa conversão como exercício.

Algoritmos de tempo pseudopolinomial

Além de ser outra aplicação útil da técnica de programação dinâmica, o Teorema 5.16 apresenta algo muito interessante – ele declara que o tempo de execução de nosso algoritmo depende do parâmetro W que, ao pé da letra, não é proporcional ao tamanho da entrada (os n itens com seus benefícios e pesos, mais o ***número*** W). Assumindo que W esteja codificado em alguma forma padrão (tal como um número binário), então ele consome apenas $O(\log W)$ *bits* para codificar W. Além disso, se W for muito grande, (digamos, $W = 2^n$), então este algoritmo de programação dinâmica poderá, na verdade, ser assintoticamente mais lento do que o método da força bruta. Assim, falando tecnicamente, este algoritmo não é de tempo polinomial, pois seu tempo de execução não é na verdade função do ***tamanho*** da entrada.

É comum se referirem a um algoritmo tal como o algoritmo de programação dinâmica para o problema da mochila como sendo um algoritmo de ***tempo pseudopolinomial***, pois seu tempo de execução depende da magnitude de um número fornecido na entrada, e não de seu tamanho de codificação. Na prática, esses algoritmos executam mais rapidamente do que o algoritmo de força bruta, mas não é correto dizer que são verdadeiros algoritmos de tempo polinomial. De fato, existe uma teoria conhecida co-

mo ***NP-completude***, discutida no Capítulo 13, que declara não ser provável que alguém encontre um algoritmo de tempo polinomial para o problema da mochila 0-1.

Em outro trecho deste livro, fornecemos aplicações adicionais para a técnica de programação dinâmica para calcular atingibilidade em um grafo direcionado (Seção 6.4.2) e para testar a similaridade entre duas *strings* (Seção 9.4).

5.4 Exercícios

Reforço

R-5.1 Faça $S = \{a,b,c,d,e,f,g\}$ ser uma coleção de objetos com valores de benefício-peso associados como segue: a:(12,4), b:(10,6), c:(8,5), d:(11,7), e:(14,3), f:(7,1), g:(9,6). Qual é a solução ótima do problema da mochila fracionada para S, assumindo que temos uma mochila que pode armazenar objetos que pesem no máximo 18? Mostre o desenvolvimento.

R-5.2 Descreva como implementar o método TaskSechedule para executar em tempo $O(n \log n)$.

R-5.3 Suponha um conjunto de tarefas especificadas por pares de tempo inicial e tempo final como $T = \{(1,2), (1,3), (1,4), (2,5), (3,7), (4,9), (5,6), (6,8), (7,9)\}$. Resolva o problema de escalonamento de tarefas para este conjunto.

R-5.4 Caracterize cada uma das seguintes equações recorrentes usando o método mestre (assumindo que $T(n) = c$ para $n < d$, para $c > 0$ e $d \geq 1$ constantes).

a. $T(n) = 2T(n/2) + \log n$
b. $T(n) = 8T(n/2) + n^2$
c. $T(n) = 16T(n/2) + (n \log n)^4$
d. $T(n) = 7T(n/3) + n$
e. $T(n) = 9T(n/3) + n^3 \log n$

R-5.5 Use o algoritmo de divisão e conquista, da Seção 5.2.2, para calcular $10110011 \cdot 10111010$ em binário. Mostre o desenvolvimento.

R-5.6 Use o algoritmo de Strassen para multiplicação de matrizes para multiplicar as matrizes:

$$X = \begin{pmatrix} 3 & 2 \\ 4 & 8 \end{pmatrix} \quad \text{e} \quad Y = \begin{pmatrix} 1 & 5 \\ 9 & 6 \end{pmatrix}.$$

R-5.7 Um número complexo $a+b\mathbf{i}$, onde $\mathbf{i} = \sqrt{-1}$, pode ser representado pelo par (a,b). Descreva um método que execute apenas três multiplicações de números reais para calcular o par (e,f) representando o produto de $a + b\mathbf{i}$ e $c + d\mathbf{i}$.

R-5.8 Matrizes booleanas são matrizes cujas entradas são 0 ou 1, e a multiplicação dessas matrizes é executada usando AND para $\cdot$ e OR para +. Suponha que sejam dadas duas matrizes booleanas aleatórias $n \times n$, A e B, cuja pro-

babilidade de uma entrada ser 1 é $1/k$. Mostre que, se k é uma constante, então existe um algoritmo para multiplicar A e B cujo tempo esperado de execução é $O(n^2)$. O que muda se k for n?

R-5.9 Qual a melhor maneira de multiplicar uma cadeia de matrizes com dimensões 10×5; 5×2; 2×20; 20×12; 12×4 e 4×60? Mostre o desenvolvimento.

R-5.10 Projete um algoritmo eficiente para o problema do produto de matrizes que resulte em uma expressão completamente parentetizada que permita multiplicar as matrizes da cadeia, usando um número mínimo de operações.

R-5.11 Resolva o Exercício R-5.1 considerando o problema da mochila 0-1.

R-5.12 Sally está hospedando um leilão pela Internet para vender n bugigangas. Ela recebe m ofertas, cada uma na forma "Eu quero k_i bugigangas por d_i dólares", para $i = 1, 2,..., m$. Caracterize seu problema de otimização como um problema da mochila. Sob que condições este pode ser visto como um problema 0-1 *versus* um problema fracional?

Criatividade

C-5.1 Um nativo australiano chamado Anatjari deseja atravessar o deserto carregando apenas uma única garrafa de água. Ele tem um mapa que marca todas as fontes de água ao longo do caminho. Assumindo que ele pode caminhar k milhas com uma garrafa de água, projete um algoritmo eficiente para determinar onde Anatjari deverá encher sua garrafa de maneira a fazer o menor número de paradas possível. Argumente por que seu algoritmo está correto.

C-5.2 Considere o problema de ***escalonamento de uma máquina*** em que temos um conjunto T de tarefas especificadas por seus tempos de início e fim como em um problema de escalonamento de tarefas, exceto pelo fato de que agora temos apenas uma máquina e queremos maximizar o número de tarefas que essa máquina pode executar. Projete um algoritmo guloso para este problema de alocação de tarefas para uma única máquina e mostre que o mesmo está correto. Qual o tempo de execução deste algoritmo?

C-5.3 Descreva um algoritmo guloso eficiente para fazer troco de um valor específico usando o menor número de moedas, assumindo que existem quatro denominações para moedas (chamadas *quarters*, *dimes*, *nickels* e *pennies*) com valores de 25, 10, 5, e 1, respectivamente. Argumente por que seu algoritmo está correto.

C-5.4 Forneça um exemplo de um conjunto de denominações de moedas que faça o algoritmo guloso de fazer troco não usar o número mínimo de moedas.

C-5.5 No problema do ***vigia da galeria de arte***, temos uma linha L que representa um longo corredor da galeria de arte. Temos também um conjunto $X = \{x_0, x_1,..., x_{n-1}\}$ de números reais que especificam as posições das pinturas nesse corredor. Suponha que um único guarda pode proteger todas as pinturas que estejam a uma distância máxima de 1 de sua posição (em ambos os

lados). Projete um algoritmo para encontrar uma posição para os guardas que use o menor número de guardas para vigiar todas as pinturas cujas posições estão em X.

C-5.6 Projete um algoritmo de divisão e conquista para encontrar o menor e o maior elemento entre n números usando não mais do que $3n/2$ comparações.

C-5.7 Dado um conjunto P de n times em algum esporte, um ***torneio round-robin**** é uma seqüência de jogos na qual cada time joga com cada um dos outros exatamente uma vez. Projete um algoritmo eficiente para organizar um torneio *round-robin* assumindo que n é uma potência de 2.

C-5.8 Seja dado um conjunto de intervalos $S = \{[a_0,b_0], [a_1,b_1],..., [a_{n-1},b_{n-1}]\}$ do intervalo [0,1], com $0 \leq a_i < b_i \leq 1$, para $i = 0,1,..., n-1$. Suponha que atribuímos um peso h_i para cada intervalo $[a_i,b_i]$ de S. O ***envelope superior*** de S é definido como sendo a lista de pares $[(x_0,c_0), (x_1,c_1), (x_2,c_2),..., (x_m,c_m), (x_{m+1},0)]$, com $x_0 = 0$ e $x_{m+1} = 1$, ordenados pelos valores de x_i, de forma que para cada subintervalo $s = [x_i,x_{i+1}]$ a altura do maior intervalo de S que contém s é c_i, para $i = 0,1,..., m$. Projete um algoritmo de tempo $O(n \log n)$ para calcular o envelope superior de S.

C-5.9 Como podemos modificar o algoritmo de programação dinâmica de um simples cálculo do melhor valor de benefício do problema da mochila 0-1 para o cálculo da atribuição que resulta neste benefício?

C-5.10 Suponha que fornecemos uma coleção $A = \{ a_1, a_2,..., a_n\}$ de n inteiros positivos que somam N. Projete um algoritmo de tempo $O(nN)$ para determinar se existe um subconjunto $B \subset A$, tal que $\sum_{a_i \in B} a_i = \sum_{a_i \in A-B} a_i$.

C-5.11 Seja P um polígono convexo (Seção 12.5.1). Uma ***triangulação*** de P é a adição das diagonais que cortam os vértices de P de maneira que cada face interior seja um triangulo. O ***peso*** de uma triangulação é a soma dos comprimentos das diagonais. Assumindo que podemos calcular os comprimentos e adicioná-los e compará-los em um tempo constante, forneça um algoritmo eficiente para calcular a triangulação com menor peso para P.

C-5.12 Uma ***gramática*** G é uma forma de gerar *strings* de caracteres "terminais" a partir de um símbolo não-terminal S pela aplicação de regras simples de substituição chamadas ***produções***. Se $B \rightarrow \beta$ for uma produção, então podemos converter uma *string* da forma $\alpha B \gamma$ no string $\alpha\beta\gamma$. Uma gramática está na ***forma normal de Chomsky*** se toda a produção for da forma $A \rightarrow BC$ ou $A \rightarrow a$, onde A, B e C são caracteres não-terminais e a é um caracter terminal. Projete um algoritmo de programação dinâmica de tempo $O(n^3)$ para determinar se a *string* $x = x_0x_1 \ldots x_{n-1}$ pode ser gerada a partir do símbolo inicial S.

C-5.13 Suponha que temos uma árvore de T enraizada no nodo n e que para cada nodo v é associado um peso $w(v)$. Um ***conjunto independente*** *de* T é um subconjunto S dos nodos de T onde nenhum nodo de S é filho ou pai de nenhum outro nodo de S. Projete um algoritmo de programação dinâmica eficiente

* N. de R. Torneio em que cada competidor joga com todos os outros.

para encontrar o conjunto de nodos independentes de T com peso máximo, onde o peso de um conjunto de nodos é simplesmente a soma dos pesos dos nodos daquele conjunto. Qual o tempo de execução de seu algoritmo?

Projetos

P-5.1 Projete e implemente um pacote para números inteiros que suporte a aritmética básica para as quatro operações.

P-5.2 Implemente um sistema para resolver problemas da mochila de forma eficiente. Seu sistema deve trabalhar tanto com o problema da mochila fracional como com o 0-1. Execute uma análise experimental para testar a eficiência de seu sistema.

Notas

O termo "algoritmo guloso" foi criado por Edmonds [64] em 1971, embora o conceito já existisse. Para maiores informações sobre o método guloso e a teoria que o suporta, conhecida como teoria dos matróides, veja o livro de Papadimitriou e Steiglitz [164].

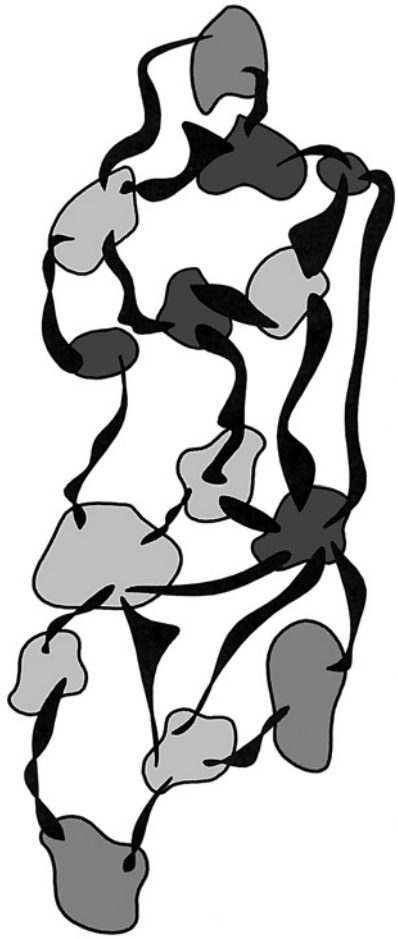
A técnica de divisão e conquista é parte do folclore do projeto de estruturas de dados e algoritmos. O método mestre para resolver recorrências de divisão e conquista tem suas origens rastreadas em um artigo de Bentley, Haken e Saxe [30]. O algoritmo de divisão e conquista para a multiplicação de inteiros grandes em tempo $O(n^{1,585})$ é normalmente atribuído aos russos Karatsuba e Ofman [111]. O algoritmo conhecido como assintoticamente mais rápido para multiplicar dois números de n dígitos é um algoritmo baseado em FFT de Schönhage e Strassen [181] que executa em tempo $O(n \log n \log \log n)$.

A programação dinâmica foi desenvolvida pela comunidade de pesquisa operacional e formalizada por Bellman [26]. A solução para o produto de matrizes encadeadas que descrevemos é atribuída a Godbole [78]. O método assintoticamente mais rápido é atribuído a Hu e Shing [101, 102]. O algoritmo de programação dinâmica para o problema da mochila é encontrado no livro de Hu [100]. Hirchsberg [95] mostra como resolver o problema da maior *substring* comum no mesmo tempo citado anteriormente, mas com ocupação linear de espaço (veja também [56]).

Parte

Algoritmos para grafos

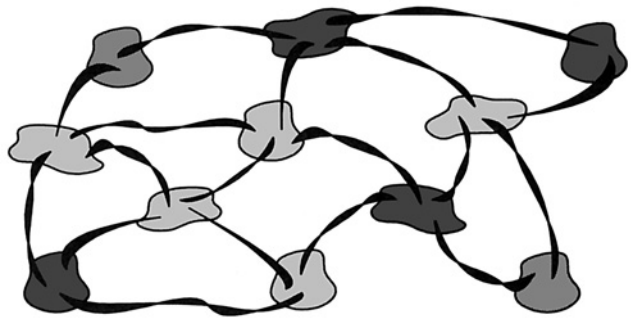
II



Capítulo

Grafos

6



6.1 O tipo abstrato de dados grafo 293

6.1.1 Métodos para grafos 297

6.2 Estruturas de dados para grafos 299

6.2.1 A lista de arestas 300

6.2.2 A lista de adjacência 302

6.2.3 A matriz de adjacência 303

6.3 Caminhamento de grafos 305

6.3.1 Caminhamento em profundidade 305

6.3.2 Componentes biconexos 310

6.3.3 Caminhamento em largura 316

6.4 Grafos dirigidos 319

6.4.1 Caminhamento em um dígrafo 320

6.4.2 Fechamento transitivo 323

6.4.3 Caminhamento em profundidade e coleta de lixo 324

6.4.4 Grafos acíclicos dirigidos 327

6.5 Exemplo em Java: caminhamento em profundidade 331

6.5.1 O padrão decorador 331

6.5.2 Um mecanismo para caminhamento em profundidade 331

6.5.3 O padrão de projeto com *templates* 333

6.6 Exercícios 337

A mitologia grega fala de um elaborado labirinto construído para abrigar o monstruoso Minotauro, parte homem e parte touro. Esse labirinto era tão complexo que nenhum animal ou homem podia escapar dele. Nenhum homem até o herói grego Teseu que, com a ajuda da filha do rei, Ariadne, decidiu implementar um dos algoritmos discutidos neste capítulo. Teseu amarrou um novelo de linha na porta do labirinto e o desenrolou à medida que caminhava pelas tortuosas passagens à procura do monstro. Teseu evidentemente conhecia o bom projeto de algoritmos, pois, após encontrar e vencer o Minotauro, ele facilmente seguiu o fio de volta à porta e até os braços de Ariadne.

Ser capaz de determinar quais objetos, como as passagens do labirinto, estão ligados a quais outros objetos pode não ser de importância tão vital quanto nesta história, mas mesmo assim é fundamental. A informação de conectividade está presente, por exemplo, em mapas, nos quais os objetos são estradas, e em tabelas de roteamento para a Internet, nas quais os objetos são computadores. A informação de conectividade está presente também nas relações de pai-filho definidas por uma árvore binária, cujos objetos são os nodos da árvore. De fato, a informação de conectividade pode ser definida para todos os tipos de relações que existem entre pares de objetos. O tópico que estudaremos neste capítulo – ***grafos*** – é portanto focado na representação e nos algoritmos para lidar eficientemente com essas relações. Ou seja, um grafo é um conjunto de objetos chamados vértices com uma coleção de conexões entre pares de vértices.

Grafos têm aplicações em vários domínios diferentes, incluindo mapas (em sistemas de informação geográfica), transportes (em conexões aéreas e rodoviárias), engenharia elétrica (em circuitos) e redes de computadores (em conexões na Internet). Como aplicações de grafos são tão comuns e diversas, foi desenvolvida uma extensa terminologia para descrever os diferentes componentes e propriedades dos grafos. Felizmente, já que a maior parte das aplicações de grafos é recente, esta terminologia é bastante intuitiva.

Começamos este capítulo revisando uma boa parte desta terminologia e apresentando o TAD grafo, incluindo algumas das suas propriedades elementares. Tendo apresentado o TAD grafo, apresentamos na Seção 6.2 as três principais estruturas de dados para representar grafos. Como nas árvores, os caminhamentos são importantes para grafos, o que discutiremos na Seção 6.3. Discutiremos grafos dirigidos na Seção 6.4, em que as relações têm uma direção e os problemas de conectividade tornam-se muito mais interessantes. Finalmente, na Seção 6.5, apresentamos um estudo de caso de pesquisa em profundidade. Esse estudo de caso envolve o uso de dois padrões de projeto em engenharia de *software* – o padrão de decoradores e o padrão de *templates* – bem como uma discussão de como a pesquisa em profundidade pode ser usada para a coleta de lixo.

6.1 O tipo abstrato de dados grafo

Visto de forma abstrata, um ***grafo*** G é simplesmente um conjunto V de ***vértices*** e uma coleção E de pares de vértices de V, chamados ***arestas***. Assim, um grafo é uma forma de representar conexões ou relações entre pares de objetos de algum conjunto V. Alguns livros usam terminologia diferente para grafos e referem-se ao que chamamos de vértices como ***nodos*** e o que chamamos de arestas como ***arcos***. Usaremos os termos "vértices" e "arestas".

Arestas em um grafo podem ser ***dirigidas*** ou ***não-dirigidas***. Uma aresta (u,v) é dita ***dirigida*** de u para v se o par (u,v) for ordenado, com u precedendo v. Uma aresta (u,v) é dita ***não-dirigida*** se o par (u,v) não for ordenado. Arestas não-dirigidas são por vezes denotadas como conjuntos $\{u,v\}$, mas para simplificar usaremos a notação de pares ordenados (u,v), notando que no caso não-dirigido (u,v) é o mesmo que (v,u). Grafos são tipicamente visualizados desenhando-se os vértices como ovais ou retângulos, e as arestas como segmentos ou curvas conectando pares de ovais ou retângulos. A seguir apresentamos alguns exemplos de grafos dirigidos e não-dirigidos.

Exemplo 6.1: *podemos visualizar colaborações entre pesquisadores de certa área construindo um grafo cujos vértices são associados com os pesquisadores e cujas arestas conectam pares de vértices associados com os pesquisadores que escreveram juntos um artigo ou livro* (*veja a Figura 6.1*). *Tais arestas são não-dirigidas porque a co-autoria é uma relação* ***simétrica****, ou seja, se A é co-autor de B, então necessariamente B é co-autor de A.*

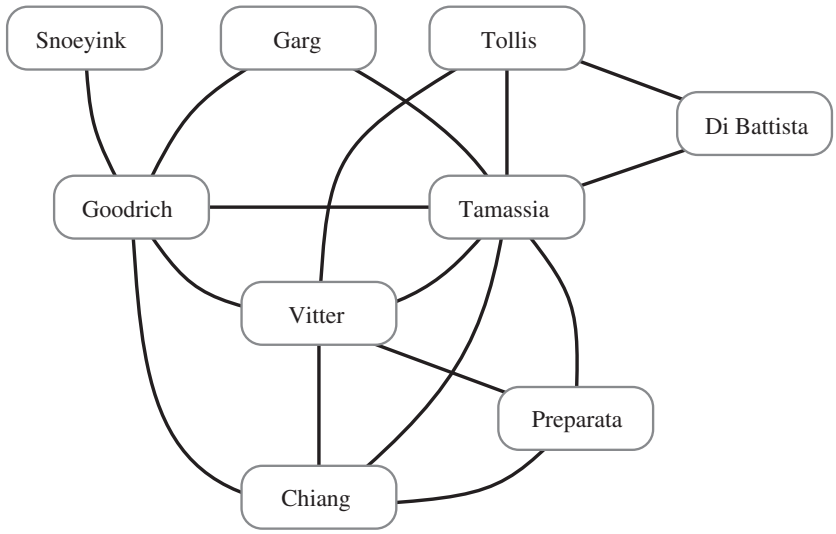


Figura 6.1 Grafo de co-autoria entre alguns autores.

Exemplo 6.2: *podemos associar a um programa orientado a objetos um grafo cujos vértices representam as classes definidas no programa e cujas arestas indicam a herança entre as classes. Existe uma aresta de um vértice v para um vértice u se a classe v estende a classe de u. Tais arestas são dirigidas porque a relação de herança só existe em uma direção* (*ou seja, ela é* ***assimétrica***).

Se todas as arestas de um grafo forem não-dirigidas, então dizemos que o grafo é um ***grafo não-dirigido***. De forma similar, um ***grafo dirigido***, ou ***dígrafo***, é um grafo onde todas as arestas são dirigidas. Um grafo que tem arestas dirigidas e não-dirigidas é chamado ***grafo misto***. Note que um grafo não-dirigido ou misto pode ser transformado em um grafo dirigido substituindo-se cada aresta não-dirigida (u,v) por um par de arestas dirigidas (u,v) e (v,u). No entanto, às vezes é útil manter grafos não-dirigidos ou mistos em sua forma original, pois esses grafos têm várias aplicações.

Exemplo 6.3: *um mapa de cidade pode ser modelado como um grafo cujos vértices são cruzamentos ou finais de ruas e cujas arestas podem ser trechos de ruas sem cruzamentos. Esse grafo tem arestas não-dirigidas, representando ruas de dois sentidos e arestas dirigidas, correspondendo a trechos de um único sentido. Assim, um grafo que representa as ruas de uma cidade é um grafo misto.*

Exemplo 6.4: *exemplos físicos de grafos estão presentes nas redes elétricas e de encanamento de um prédio. Tais redes podem ser modeladas como grafos em que cada conector, junção ou saída é visto como um vértice, e cada trecho não-interrompido de fiação ou cano é visto como uma aresta. Tais grafos são na verdade componentes de grafos muito maiores – as redes locais de distribuição de energia e de água. Dependendo dos aspectos específicos dos grafos em que estamos interessados, podemos considerar suas arestas como dirigidas ou não-dirigidas, pois, em princípio, a água pode fluir em um cano nas duas direções, assim como a corrente em um fio.*

Os dois vértices conectados por uma aresta são chamados de ***vértices finais*** da aresta. Os vértices finais de uma aresta também são conhecidos como ***pontos finais*** da aresta. Se uma aresta for dirigida, seu primeiro ponto final é sua ***origem*** e o outro é seu ***destino***.

Dois vértices são ditos ***adjacentes*** se forem pontos finais da mesma aresta. Uma aresta é dita ***incidente*** a um vértice se o vértice é um dos pontos finais da aresta. As arestas incidentes ***de*** um vértice são as arestas dirigidas cuja origem é aquele vértice. As arestas incidentes ***em*** um vértice são as arestas dirigidas cujo destino é aquele vértice. O ***grau*** de um vértice v, denotado $deg(v)$, é o número de vértices incidentes a v. O ***grau de entrada*** e o ***grau de saída*** de um vértice v são os números de arestas incidentes em e de v, respectivamente, e são denotados $indeg(v)$ e $outdeg(v)$.

Exemplo 6.5: *podemos estudar o transporte aéreo construindo um grafo G chamado* ***malha aérea****, cujos vértices são associados a aeroportos e cujas arestas são associadas com vôos (veja a Figura 6.2). No grafo G, as arestas são dirigidas porque um dado vôo tem uma direção específica (do aeroporto de origem ao aeroporto de destino). Os pontos finais de uma aresta e em G correspondem respectivamente à origem e ao destino do vôo correspondente a e. Dois aeroportos são adjacentes em G se existir um vôo entre eles, e uma aresta e é incidente a um vértice v de G se o vôo representado por e sair do aeroporto ou chegar ao aeroporto representado por v. As arestas incidentes de um vértice v correspondem aos vôos que saem do aeroporto de v, enquanto as arestas incidentes em v correspondem aos vôos que chegam. Finalmente, o grau de entrada de um vértice v de G corresponde ao número de vôos que chegam ao aeroporto de v, enquanto o grau de saída representa o número de vôos que saem.*

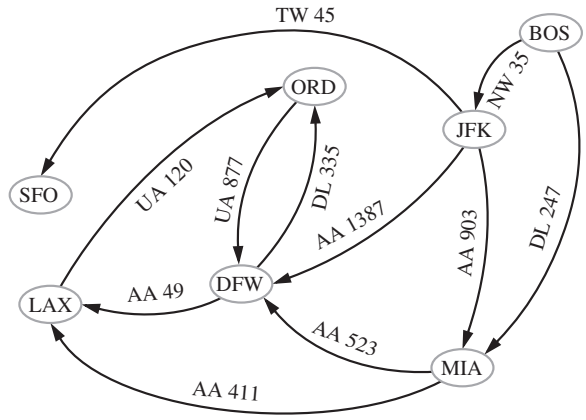


Figura 6.2 Exemplo de um grafo dirigido representando uma malha aérea. Os pontos finais da aresta UA120 são LAX e ORD; portanto, LAX e ORD são adjacentes. O grau de entrada de DFW é 3 e o grau de saída de DFW é 2.

A definição de grafo agrupa arestas como uma ***coleção***, não como um ***conjunto***, permitindo que duas arestas não-dirigidas tenham os mesmos pontos finais e que duas arestas dirigidas tenham a mesma origem e o mesmo destino. Tais arestas são chamadas ***arestas paralelas*** ou ***arestas múltiplas***. As arestas paralelas podem estar em uma malha aérea (Exemplo 6.5) e, neste caso, múltiplas arestas entre o mesmo par de vértices indica vôos diferentes operando na mesma rota em diferentes horas do dia. Outro tipo especial de aresta é o que conecta um vértice consigo mesmo. Assim, dizemos que uma aresta (dirigida ou não-dirigida) forma um ***laço*** se seus pontos finais coincidirem. Um laço pode ocorrer em um grafo associado com um mapa urbano (Exemplo 6.3), no qual corresponderia a uma "circular" (uma rua circular que retorna a seu ponto de início).

Com poucas exceções, como as mencionadas acima, grafos não têm arestas paralelas ou laços. Estes grafos são ditos ***simples***. Assim, podemos geralmente dizer que as arestas de um grafo simples são um ***conjunto*** de pares de vértices (e não uma coleção). Neste capítulo, assumiremos que um grafo é simples a não ser que seja especificado de outra forma. Isso simplifica a apresentação de estruturas de dados e algoritmos para grafos. Estender os resultados deste capítulo para grafos generalizados que admitem laços e arestas paralelas não é difícil, embora os detalhes possam ser tediosos.

Exploraremos, nos teoremas a seguir, algumas propriedades importantes dos graus e do número de vértices de um grafo. Essas propriedades relacionam o número de arestas e de vértices um ao outro e aos graus dos vértices de um grafo.

Teorema 6.6: *se G é um grafo com m arestas, então*

$$\sum_{v \in G} deg(v) = 2m.$$

Prova: uma aresta (u,v) é contada duas vezes na soma acima: uma por seu ponto final u e outra por seu ponto final v. Assim, a contribuição total das arestas para os graus dos vértices é de duas vezes o número de arestas. ■

Teorema 6.7: *se G é um grafo dirigido com m arestas, então*

$$\sum_{v \in G} indeg(v) = \sum_{v \in G} outdeg(v) = m.$$

Prova: em um grafo dirigido, uma aresta (u,v) contribui com uma unidade para o grau de saída de sua origem u e uma unidade para o grau de entrada de seu destino v. Assim, a contribuição total das arestas para os graus de saída dos vértices é igual ao número de arestas e similarmente para os graus de entrada. ■

Teorema 6.8: *seja G um grafo simples com n vértices e m arestas. Se G é não-dirigido, então* $m \leq n(n-1)/2$, *e se G é dirigido, então* $m \leq n(n-1)$.

Prova: suponha que G seja não-dirigido. Como duas arestas não podem ter os mesmos pontos de saída e de chegada e não há laços, o grau máximo de um vértice em G é $n-1$ neste caso. Assim, pelo Teorema 6.6, temos $2m \leq n(n-1)$. Agora vamos supor que G seja dirigido. Como duas arestas não podem ter os mesmos pontos de saída e de chegada e não há laços, o grau máximo de entrada de um vértice em G é $n-1$ neste caso. Assim, pelo Teorema 6.7, $m \leq n(n-1)$. ■

Colocado de outra forma, o Teorema 6.8 diz que um grafo simples com n vértices tem $O(n^2)$ arestas.

Um ***caminho*** em um grafo é uma seqüência alternada de vértices e arestas que se inicia em um vértice e termina em um vértice de tal forma que cada aresta é incidente a seu vértices antecessor e sucessor. Um ***ciclo*** é um caminho em que os vértices de início e fim são os mesmos. Dizemos que um caminho é ***simples*** se cada vértice no caminho for distinto; dizemos que um ciclo é ***simples*** se cada vértice no ciclo é distinto, exceto pelo primeiro e último. Um ***caminho dirigido*** é um caminho em que todas as arestas são dirigidas e percorridas em sua direção. Um ***ciclo dirigido*** é definido de forma similar.

Exemplo 6.9: *dado um grafo G representando o mapa de uma cidade* (*veja o Exemplo* 6.3)*, podemos representar um casal dirigindo de casa até um restaurante como um caminho em G. Se eles souberem o caminho e não passarem acidentalmente pelo mesmo cruzamento duas vezes, então eles passam por um caminho simples em G. Podemos representar o caminho completo do casal, de casa ao restaurante e de volta, como um ciclo. Se eles voltarem para casa por um caminho completamente diferente do usado para chegar ao restaurante, sem passar em um mesmo cruzamento, então toda a viagem de ida e volta é um ciclo simples. Finalmente, se eles só passam por ruas de mão-única, então podemos representar sua saída como um ciclo dirigido.*

Um ***subgrafo*** de um grafo G é um grafo H cujos vértices e arestas são respectivamente subconjuntos dos vértices e arestas de G. Um grafo é ***conexo*** se para quaisquer dois vértices existir um caminho entre eles. Se um grafo G não é conexo, seus subgrafos conexos maximais são chamados de ***componentes conexos*** de G. Uma ***floresta*** é um grafo sem ciclos. Uma ***árvore*** é uma floresta conexa, ou seja, um grafo conexo sem ciclos.

Note que essa definição de uma árvore é um pouco diferente da definição fornecida na Seção 2.3. Ou seja, no contexto dos grafos, uma árvore não tem raiz. Sempre que houver ambigüidade, as árvores da Seção 2.3 são chamadas de ***árvores com raiz***, enquanto que as árvores deste capítulo são chamadas de ***árvores livres***. Os componentes

conexos de uma floresta são árvores (livres). Uma ***árvore de cobertura*** de um grafo é um subgrafo de cobertura que é uma árvore (livre).

Exemplo 6.10: *talvez o grafo mais popular atualmente seja a Internet, que pode ser vista como um grafo cujos vértices são computadores e cujas arestas (não-dirigidas) são conexões de comunicação entre pares de computadores na Internet. Os computadores e as conexões em um único domínio como* wiley.com *formam um subgrafo da Internet. Se esse subgrafo é conexo, então dois usuários em computadores deste domínio podem mandar mensagens um ao outro sem que os pacotes de informação deixem o domínio. Suponha que as arestas desse subgrafo formem uma árvore de cobertura. Isso implica que, se uma única conexão for desfeita (por exemplo, porque alguém desliga o cabo de rede de um computador deste domínio), então o subgrafo não será mais conexo.*

Existem uma série de propriedades simples de árvores, florestas e grafos conexos. Exploraremos algumas delas na proposição a seguir.

Teorema 6.11: seja G um grafo não-dirigido com n vértices e m arestas. Então temos:

- Se G é conexo, então $m \geq n - 1$.
- Se G é uma árvore, então $m = n - 1$.
- Se G é uma floresta, então $m \leq n - 1$.

Deixamos a justificativa desta proposição como exercício (C-6.1).

6.1.1 Métodos para grafos

Como um tipo abstrato de dados, um grafo é um contêiner posicional de elementos que são armazenados nos vértices e arestas do grafo. Ou seja, as ***posições*** em um grafo são seus vértices e arestas. Portanto, podemos armazenar elementos em um grafo tanto em seus vértices quanto em suas arestas (ou ambos). Em termos de uma implementação orientada a objetos, isso implica que podemos definir tipos abstratos de dados Vertex* e Edge** que estendem o TAD Position. Lembre da Seção 2.2.2, que uma posição tem um método element( ) que retorna o elemento armazenado na posição. Também usamos iteradores especializados para vértices e arestas que nos permitem enumerar iterativamente uma coleção de vértices ou de arestas respectivamente. Já que um grafo é um contêiner posicional, o TAD grafo suporta os métodos size( ), isEmpty( ), elements( ), positions( ), replaceElement(p, o) e swapElements(p, q), onde p e q denotam posições e o denota um objeto, ou seja, um elemento.

Grafos são um tipo abstrato de dados muito mais rico do que os que discutimos nos capítulos anteriores. Sua riqueza deriva principalmente dos diferentes tipos de objetos que ajudam a definir um grafo, como vértices e arestas. Para apresentar os métodos para o TAD grafo da forma mais organizada possível, dividimos os métodos em três categorias principais: métodos gerais, métodos que lidam com vértices dirigidos e métodos para atualizar e modificar grafos. Além disso, para simplificar a apresentação, indicamos com v um vértice, com e uma aresta e com o um objeto (elemento) armazenado em um vértice ou aresta. Também não discutiremos condições de erro que possam ocorrer.

* N. de T. Vértice.
** N. de T. Aresta.

Métodos gerais

Começamos descrevendo os métodos fundamentais para grafos, que ignoram a direção das arestas. Cada um dos métodos a seguir retorna informação global sobre um grafo G:

numVertices(): retorna o número de vértices em G.

numEdges(): retorna o número de arestas em G.

vertices(): retorna um iterador para os vértices em G.

edges(): retorna um iterador para as arestas em G.

Diferentemente de uma árvore (que tem uma raiz), um grafo não tem um vértice especial. Portanto, temos um método que retorna um vértice arbitrário de um grafo:

aVertex(): retorna um vértice de G.

Os métodos de acesso a seguir recebem vértices e arestas como argumentos:

degree(v): retorna o grau de v.

adjacentVertices(v): retorna um iterador para os vértices adjacentes a v.

incidentEdges(v): retorna um iterador para as arestas incidentes a v.

endVertices(e): retorna um arranjo de tamanho 2 com os vértices finais de e.

opposite(v, e): retorna o ponto final da aresta e diferente de v.

areAdjacent(v, w): retorna se v e w são adjacentes.

Métodos para vértices dirigidos

Quando permitimos que algumas ou todas as arestas de um grafo sejam dirigidas, então há vários métodos adicionais que devem ser incluídos no TAD grafo. Começamos com métodos que lidam especificamente com arestas dirigidas:

directedEdges(v): retorna um iterador para todas as arestas dirigidas.

undirectedEdges(v): retorna um iterador para todas as arestas não-dirigidas.

destination(e): retorna o destino da aresta dirigida e.

origin(e): retorna a origem da aresta dirigida e.

isDirected(e): retorna *true* se e somente se a aresta e é dirigida.

Além desses métodos, a existência de arestas dirigidas requer que tenhamos maneiras de relacionar vértices e arestas em termos de direções:

inDegree(v): retorna o grau de entrada de v.

outDegree(v): retorna o grau de saída de v.

inIncidentEdges(v): retorna um iterador para todas as arestas chegando a v.

outIncidentEdges(v): retorna um iterador para todas as arestas saindo de v.

inAdjacentVertices(v): retorna um iterador para todos vértices adjacentes a v por arestas que chegam a v.

outAdjacentVertices(v): retorna um iterador para todos vértices adjacentes a v por arestas que saem de v.

Métodos para modificar grafos

Também podemos ter métodos de motificação que adicionam ou deletam arestas e vértices:

insertEdge(v,w,o): insere e retorna uma aresta não-dirigida entre os vértices v e w, armazenando o objeto o.

insertDirectedEdge(v,w,o): insere e retorna uma aresta dirigida entre os vértices v e w, armazenando o objeto o.

insertVertex(o): insere e retorna um novo vértice (isolado) armazenando o objeto o.

removeVertex(v): remove o vértice v e todas as suas arestas incidentes.

removeEdge(e): remove a aresta e.

makeUndirected(e): torna não-dirigida a aresta e.

reverseDirection(e): reverte a direção da aresta dirigida e.

setDirectionFrom(e,v): torna a aresta e dirigida, saindo do vértice v.

setDirectionTo(e,v): torna a aresta e dirigida, chegando ao vértice v.

Certamente existem muitos métodos no TAD grafo. O grande número de métodos é de certa forma inevitável, pois grafos são estruturas muito ricas. Grafos suportam dois tipos de posições – vértices e arestas – e ainda permitem que as arestas sejam dirigidas ou não-dirigidas. Precisamos ter métodos diferentes para acessar e atualizar todas essas posições diferentes, bem como lidar com as relações que podem existir entre essas diferentes posições.

6.2 Estruturas de dados para grafos

Existem várias maneiras de implementar o TAD grafo com uma estrutura de dados concreta. Nesta seção, discutiremos três alternativas populares, geralmente referidas como ***lista de arestas***, ***lista de adjacência*** e ***matriz de adjacência***. Nas três representações, usamos um contêiner (uma lista ou vetor, por exemplo) para armazenar os vértices do grafo. Em relação às arestas, existe uma diferença fundamental entre as duas primeiras estruturas e a terceira. A lista de arestas e a lista de adjacência armazenam apenas as arestas realmente presentes no grafo, enquanto a matriz de adjacência armazena uma posição para cada par de vértices (se houver uma aresta entre eles ou não). Como explicaremos nesta seção, essa diferença explica que, para um grafo G com n vértices e m arestas, uma lista de arestas ou lista de adjacência usam espaço $O(m + n)$, enquanto uma matriz de adjacência usa espaço $O(n^2)$.

6.2.1 A lista de arestas

A estrutura da ***lista de arestas*** é possivelmente a mais simples, embora não a mais eficiente representação de um grafo G. Nesta representação, um vértice v de G armazenando um elemento o é explicitamente representado por um objeto vértice. Todos esses objetos vértices são armazenados em um contêiner V, que pode ser tipicamente uma lista, vetor ou dicionário. Se representarmos V como um vetor, por exemplo, então pensaríamos naturalmente nos vértices como sendo numerados. Se representarmos V como um dicionário, por outro lado, então pensaríamos nos vértices como sendo identificados por uma chave associada a eles. Note que os elementos do contêiner V são os vértices do grafo G.

Objetos vértice

O objeto vértice para um vértice v armazenando o elemento o tem variáveis instanciadas para:

- uma referência a o;
- contadores para o número de arestas incidentes não-dirigidas, arestas incidentes dirigidas que chegam e arestas incidentes dirigidas que saem;
- uma referência para a posição (ou localizador) do objeto vértice no contêiner V.

A principal característica da lista de arestas não é a maneira como ela representa os vértices, mas como ela representa as arestas. Nesta estrutura, uma aresta e de G armazenando um elemento o é explicitamente representada por um objeto aresta. Os objetos arestas são armazenados em um contêiner E, que seria tipicamente uma lista, vetor ou dicionário (possivelmente suportando localizadores).

Objetos aresta

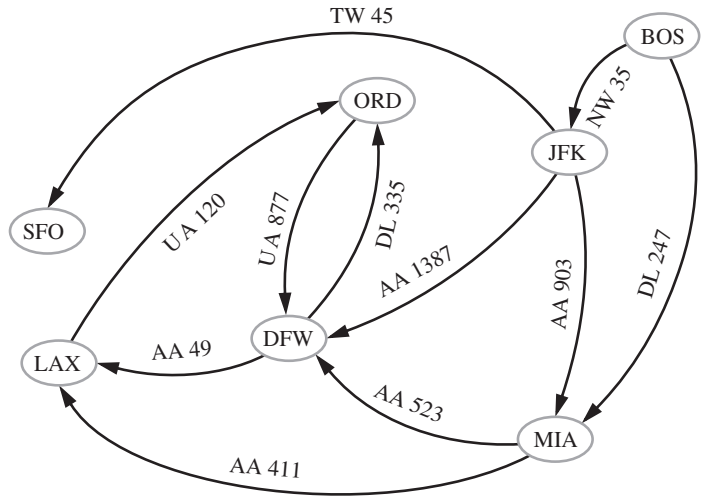
O objeto aresta para uma aresta e armazenando o objeto o tem variáveis instanciadas para:

- uma referência a o;
- um booleano indicando se e é dirigido ou não-dirigido;
- referências para os objetos vértice em V associados com os pontos finais de e (se e é não-dirigida) ou para os vértices de origem e destino (se e é dirigida);
- uma referência para a posição (ou localizador) do objeto aresta no contêiner E.

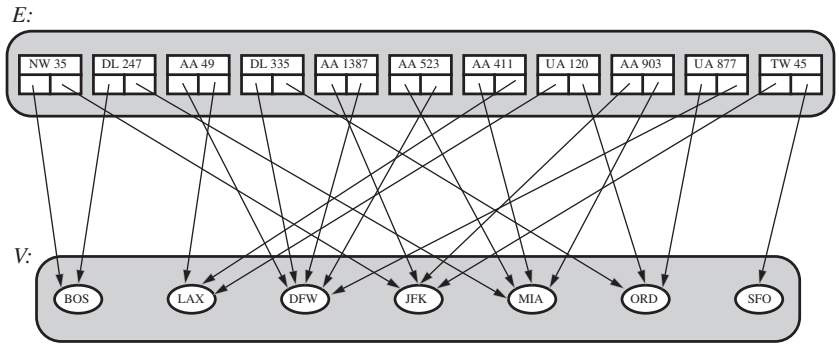
Ilustramos uma lista de arestas para um grafo dirigido G na Figura 6.3.

A lista de arestas

A razão pela qual esta estrutura é chamada de lista de arestas é porque a implementação mais simples e comum do contêiner E é uma lista. Mesmo assim, para poder procurar convenientemente objetos específicos associados a uma aresta, podemos desejar implementar E com um dicionário, apesar de continuar chamando a estrutura de "lista de arestas". Também podemos desejar implementar o contêiner V como um dicionário pela mesma razão. Ainda assim, mantendo a tradição, chamamos a estrutura de "lista de arestas".



(a)



(b)

Figura 6.3 (a) Um grafo dirigido G; (b) representação esquemática da lista de arestas para G. Não mostramos os campos a seguir dos objetos vértice: três contadores para as arestas incidentes e a referência para a posição (ou localizador) do objeto vértice no contêiner V. Também não mostramos os campos a seguir dos objetos aresta: o indicador booleano de direção e a referência para a posição (ou localizador) do objeto aresta no contêiner E. Finalmente, visualizamos os elementos armazenados nos objetos vértice e aresta com os nomes dos elementos, em vez de referências reais aos elementos.

A característica principal da lista de arestas é que ela provê acesso direto das arestas para os vértices aos quais elas são incidentes. Isso permite-nos definir algoritmos simples para implementar os diferentes métodos baseados em arestas para o TAD grafo (por exemplo, os métodos endVertices, origin e destination). Podemos simplesmente acessar os componentes apropriados do objeto aresta para implementar cada método.

Mesmo assim, a operação inversa – acessar as arestas incidentes a um vértice – requer uma inspeção exaustiva de todos os objetos aresta no contêiner E. Ou seja, para determinar quais arestas são incidentes a um vértice v, devemos examinar cada aresta na lista de arestas e verificar se ela é incidente a v. Assim, por exemplo, o método incidentEdges(v) tem tempo proporcional ao número de arestas do grafo, não ao grau do

vértice v. De fato, mesmo verificar se dois vértices v e w são adjacentes com o método areAdjacent(v,w) requer que procuremos toda a lista de arestas procurando pela aresta (v,w) ou (w,v). Além disso, já que remover um vértice requer remover também suas arestas incidentes, o método removeVertex também requer uma busca completa na lista de arestas E.

Desempenho

Entre as principais características de desempenho das listas de arestas, assumindo que V e E são realizados com listas duplamente encadeadas, estão as seguintes:

- Os métodos numVertices( ), numEdges( ) e size( ) são implementados em tempo $O(1)$ acessando informações de tamanho das listas V e/ou E.
- Os contadores armazenados com cada objeto vértice permitem-nos realizar em tempo constante os métodos degree, inDegree e outDegree.
- Os métodos vertices( ) e edges( ) são implementados retornando um iterador para as listas de vértices e de arestas, respectivamente. Da mesma forma, implementamos iteradores directedEdges( ) e undirectedEdges( ) expandindo um iterador para a lista E e retornando apenas as arestas que são do tipo correto.
- Já que os contêineres V e E são listas implementadas com uma lista duplamente encadeada, podemos inserir vértices e inserir e remover arestas em tempo $O(1)$.
- Os métodos incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices e areAdjacent custam tempo $O(m)$, pois para determinar quais arestas são incidentes a um vértice v temos de examinar todas as arestas.
- O método removeVertex(v) custa tempo $O(m)$, pois requer que examinemos todas as arestas para encontrar e remover incidentes a v.

6.2.2 A lista de adjacência

A representação por lista de arestas é simples mas tem suas limitações, pois muitos métodos que deveriam ser rápidos para vértices individuais precisam examinar toda a lista de arestas para funcionar corretamente. A ***lista de adjacência*** para um grafo G estende a lista de arestas, adicionando informação extra que suporta acesso direto aos vértices incidentes (e, portanto, aos vértices adjacentes) a cada vértice. Enquanto a lista de arestas vê a relação de incidência entre arestas e vértices apenas do ponto de vista das arestas, a lista de adjacência considera a relação pelos dois pontos de vista. Essa abordagem simétrica permite-nos usar a lista de adjacência para implementar vários métodos para vértices do TAD grafo muito mais rapidamente do que seria possível com a lista de arestas, mesmo que as duas representações usem espaço proporcional ao número de vértices e arestas do grafo. A lista de adjacência inclui todos os componentes estruturais da lista de arestas mais os seguintes:

- O objeto vértice v contém uma referência a um contêiner $I(v)$, chamado ***contêiner de incidência***, que armazena referências para as arestas incidentes em v. Se arestas dirigidas são permitidas, então dividimos $I(v)$ em $I_{in}(v)$, $I_{out}(v)$ e

$I_{un}(v)$, que armazenam respectivamente as arestas incidentes em, incidentes de e não-dirigidas de v.

- O objeto aresta para uma aresta (u,v) contém referências para as posições (ou localizadores) da aresta nos contêiners de incidência $I(u)$ e $I(v)$.

A lista de adjacência

Tradicionalmente, o contêiner de incidência $I(v)$ para um vértice v é realizado através de uma lista e por isso chamamos esta forma de representar o grafo de ***lista de adjacência***. Contudo, pode haver contextos em que desejamos representar um contêiner de incidência $I(v)$ como, digamos, um dicionário ou uma fila de prioridade; portanto, vamos continuar pensando em $I(v)$ como um contêiner genérico de objetos arestas. Se desejarmos suportar uma representação de grafos que pode representar um grafo G potencialmente contendo arestas dirigidas e não-dirigidas, então precisamos, para cada vértice, de três contêineres de incidência $I_{in}(v)$, $I_{out}(v)$ e $I_{un}(v)$, que armazenam respectivamente as arestas dirigidas incidentes em v, as arestas dirigidas incidentes de v e as arestas não-dirigidas incidentes a v.

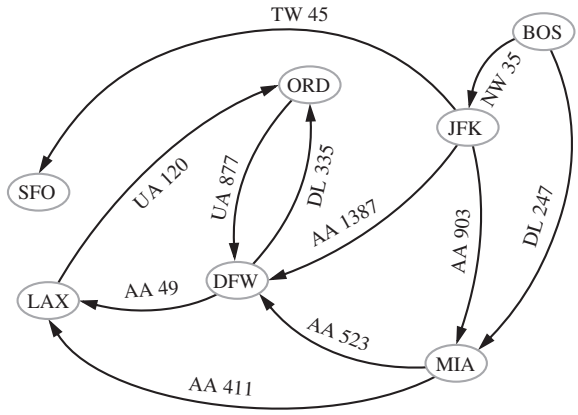
A lista de adjacência provê acesso direto tanto das arestas para os vértices quanto dos vértices para suas arestas incidentes. Ser capaz de prover acesso entre vértices e arestas em ambas as direções permite-nos acelerar o desempenho de uma série de métodos para grafos se usarmos uma lista de adjacência no lugar de uma lista de arestas. Ilustramos a lista de adjacência de um grafo dirigido na Figura 6.4. Para um vértice v, o espaço usado pelo contêiner de incidência de v é proporcional ao grau de v, ou seja, é $O(deg(v))$. Assim, pelo Teorema 6.6, o espaço necessário para a lista de adjacência é $O(n + m)$.

A lista de adjacência tem desempenho equivalente à lista de arestas, porém provê melhores tempos de execução para os seguintes métodos:

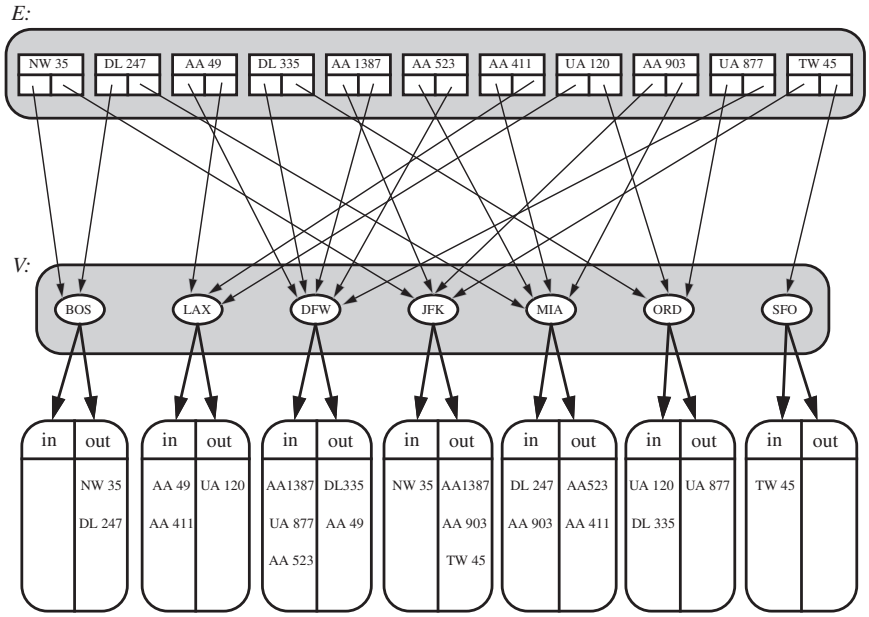
- Os métodos que retornam iteradores para as arestas incidentes ou vértices adjacentes a um vértice v podem ser construídos em tempo proporcional a seu tamanho, ou seja, em tempo $O(deg(v))$.
- O método areAdjacent(u,v) pode ser realizado inspecionando-se o contêiner de incidência de u ou o de v. Escolhendo o menor dos dois, temos o tempo de execução $O(\min\{deg(u),deg(v)\})$.
- O método removeVertex(v) requer que se use incidentEdges(v) para identificar as arestas a serem removidas em conseqüência da operação. As $deg(v)$ operações de remoção seguintes custam tempo $O(1)$ cada.

6.2.3 A matriz de adjacência

Como a lista de adjacência, a representação de um grafo por ***matriz de adjacência*** estende a estrutura de armazenamento das arestas com um componente adicional. Neste caso, aumentamos a lista de arestas com uma matriz (um arranjo de duas dimensões) A que permite que determinemos adjacências entre pares de vértices em tempo constante. Como veremos, atingir esse desempenho tem um preço no tocante ao espaço usado para a estrutura de dados.



(a)



(b)

Figura 6.4 (a) Um grafo dirigido G; (b) representação esquemática da lista de adjacência de G. Como na Figura 6.3, visualizamos os elementos dos contêineres com nomes em vez de objetos. Também mostramos apenas os contêiners de incidência para arestas dirigidas, pois não há arestas não-dirigidas neste grafo.

Na matriz de adjacência, consideramos os vértices como sendo os inteiros no conjunto $\{0, 1,..., n - 1\}$ e as arestas como sendo pares destes inteiros. Representamos o grafo G em um arranjo de duas dimensões $A_{n \times n}$, com $A[i, j]$ armazenando uma referência à aresta (i, j) se ela existir. Se ela não existir, então $A[i, j]$ é nulo.

Especificamente, a representação por matriz de adjacência estende a lista de arestas da maneira a seguir:

- Um objeto vértice v também armazena uma chave inteira única entre 0 e $n - 1$, chamada de ***índice*** de v. Para simplificar a discussão, podemos nos referir ao vértice com índice i simplesmente como "vértice i".

- Mantemos um arranjo de duas dimensões $A_{n \times n}$ tal que a célula $A[i, j]$ contém uma referência para o objeto aresta e que vai do vértice com índice i ao vértice com índice j, se tal aresta existir. Se a aresta e conectando os vértices i e j for não-dirigida, então armazenamos referências para e em $A[i, j]$ e $A[j, i]$. Se não houver aresta do vértice i ao vértice j, então $A[i, j]$ faz referência ao objeto nulo (ou outro indicador de que esta célula não está associada a nenhuma aresta).

Usando uma matriz de adjacência A podemos realizar o método areAdjacent(v, w) em tempo $O(1)$. Alcançamos esse desempenho acessando os vértices v e w para determinar seus respectivos índices i e j e então testando se a célula $A[i,j]$ contém uma referência a um objeto nulo. Esse ganho em desempenho é compensado por uma acréscimo no espaço usado, que é agora $O(n^2)$, e no tempo de execução de outros métodos. Por exemplo, os métodos como incidentEdges e adjacentVertices agora requerem que examinemos uma linha ou coluna inteira da matriz A, o que custa tempo $O(n)$. A lista de adjacência é superior à matriz de adjacência em espaço, e superior em tempo para todos os métodos exceto o método areAdjacent.

Historicamente, a matriz de adjacência foi a primeira representação usada para grafos, com a matriz de adjacência sendo definida como uma matriz booleana, como abaixo:

$$A[i, j] = \begin{cases} 1 & \text{se } (i, j) \text{ for uma aresta} \\ 0 & \text{caso contrário} \end{cases}$$

Assim, a matriz de adjacência tem um apelo natural como estrutura matemática (por exemplo, um grafo não-dirigido tem uma matriz de adjacência simétrica). Nossa definição de uma matriz de adjacência atualiza essa perspectiva histórica para um ambiente orientado a objetos. A maior parte dos algoritmos de grafos que examinamos serão mais eficientes quando agindo sobre um grafo armazenado usando uma lista de adjacência. Em alguns casos, no entanto, a situação pode se balancear dependendo de quantas arestas existem no grafo.

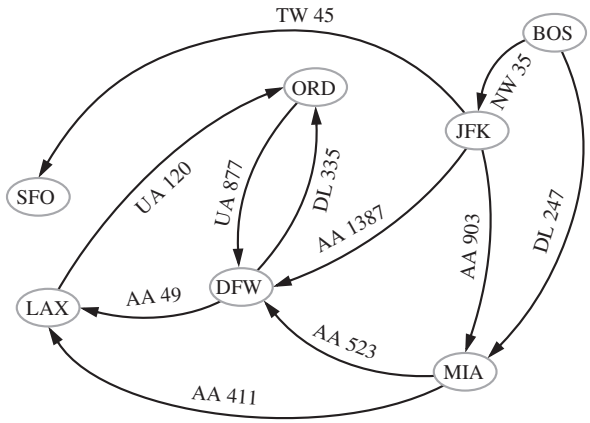
Ilustramos um exemplo de lista de adjacência na Figura 6.5.

6.3 Caminhamento de grafos

Um ***caminhamento*** é um procedimento sistemático para explorar um grafo examinando todos os seus vértices e arestas. Por exemplo, uma ***spider*** ou ***crawler*** para a Web que é a parte de coleta de dados do mecanismo de busca, deve explorar um grafo composto de documentos de hipertexto examinando seus vértices, que são os documentos, e suas arestas, que são os *links* entre os documentos. Um caminhamento é eficiente se ele visita todos os vértices e as arestas em tempo proporcional a seu número, ou seja, em tempo linear.

6.3.1 Caminhamento em profundidade

O primeiro algoritmo de caminhamento que consideramos é o ***caminhamento em profundidade*** (DFS – *depth-first search*) em um grafo não-dirigido. O caminhamento em profundidade é útil em uma variedade de computações com grafos, incluindo encontrar um caminho de um vértice a outro, determinar se um grafo é conexo ou não e achar uma árvore de cobertura de um grafo conexo.



(a)

0	1	2	3	4	5	6
BOS	DFW	JFK	LAX	MIA	ORD	SFO

(b)

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	∅	∅	NW 35	∅	DL 247	∅	∅
1	∅	∅	∅	AA 49	∅	DL 335	∅
2	∅	AA 1387	∅	∅	AA 903	∅	TW 45
3	∅	∅	∅	∅	∅	UA 120	∅
4	∅	AA 523	∅	AA 411	∅	∅	∅
5	∅	UA 877	∅	∅	∅	∅	∅
6	∅	∅	∅	∅	∅	∅	∅

(c)

Figura 6.5 Representação esquemática da matriz de adjacência: (a) um grafo dirigido G; (b) uma enumeração dos vértices em G; (c) a matriz de adjacência A para G.

Caminhamento de um grafo com *backtracking*

O caminhamento em profundidade em um grafo não-dirigido G é análogo a caminhar em um labirinto com um fio e uma lata de tinta, sem se perder. Começamos em um vértice específico s em G, o qual inicializamos amarrando nosso fio a s e pintando s pa-

ra marcá-lo como "visitado". O vértice s é agora nosso vértice corrente – vamos chamar o vértice corrente de u daqui para a frente. Percorremos G considerando uma aresta (arbitrária) (u,v) incidente ao vértice corrente u. Se a aresta (u,v) leva a um vértice v já visitado (ou seja, pintado), retornamos imediatamente ao vértice u. Se, por outro lado, (u,v) leva a um vértice v não visitado, então desenrolamos um pouco de nosso fio e vamos para v. Pintamos v como "visitado" e fazemos com que ele passe a ser o vértice corrente, repetindo a operação. Em um dado momento, iremos a um local sem saída, ou seja, o vértice corrente u tal que todas as arestas incidentes em u levam a vértices já visitados. Assim, tomar uma aresta incidente em u fará com que retornemos a u. Para sair deste impasse, enrolamos um pouco do nosso fio, retornando pela aresta que nos levou a u até um vértice já visitado v. Fazemos de v nosso vértice corrente e repetimos a operação acima para quaisquer arestas incidentes em v que não tenhamos explorado antes. Se todas as arestas incidentes em v conduzirem a vértices já visitados, então enrolamos mais um pouco do fio e retornamos ao vértice anterior a v no caminho que percorremos e repetimos a operação naquele vértice. Assim, continuamos a retornar ao longo do caminho que já percorremos até achar um vértice que ainda tem uma aresta não explorada, onde seguimos esta aresta e continuamos o caminhamento. O processo termina quando nosso retorno nos traz de volta ao vértice inicial s e não há mais arestas inexploradas incidentes a s. Esse processo simples percorre as arestas de G de uma forma elegante e sistemática (veja a Figura 6.6).

Visualização do caminhamento em profundidade

Podemos visualizar um caminhamento em profundidade que orienta as arestas pela direção em que são exploradas durante o caminhamento distinguindo as arestas usadas para descobrir novos vértices, chamadas ***arestas de descoberta*** ou ***aresta de árvore***, daquelas que levam a vértices já visitados, chamadas ***arestas de retorno*** (veja a Figura 6.6f). Na analogia acima, arestas de descoberta são as arestas onde desenrolamos nosso fio quando as percorremos, e as arestas de retorno são as arestas onde retornamos imediatamente sem desenrolar o fio. As arestas de descoberta formam uma árvore de cobertura do componente conexo do vértice de início s, chamada de ***árvore DFS***. Chamamos as arestas que não estão nesta árvore de "arestas de retorno", porque se assumirmos que a árvore tem o vértice inicial como raiz então cada uma destas arestas leva de um vértice na árvore a um de seus ancestrais na árvore.

Caminhamento em profundidade recursivo

O pseudocódigo do caminhamento em profundidade iniciando em um vértice v segue nossa analogia do fio e da tinta. Usamos a recursão para implementar a analogia do fio e assumimos que temos um mecanismo (a analogia da tinta) para determinar se um vértice ou aresta já foi explorado e para rotular as arestas como sendo de descoberta ou de retorno. Uma descrição em pseudocódigo do algoritmo recursivo de caminhamento em profundidade é mostrada no Algoritmo 6.7.

Existem várias observações que podemos fazer sobre o algoritmo de caminhamento em profundidade, das quais muitas derivam do fato de que o algoritmo divide as arestas do grafo não-dirigido G em dois grupos: as arestas de descoberta e as arestas de retorno. Por exemplo, já que arestas de retorno sempre conectam um vértice v a um vértice previamente visitado u, cada aresta de retorno implica um ciclo em G, consistindo nas arestas de descoberta de u a v mais a aresta de retorno (u,v).

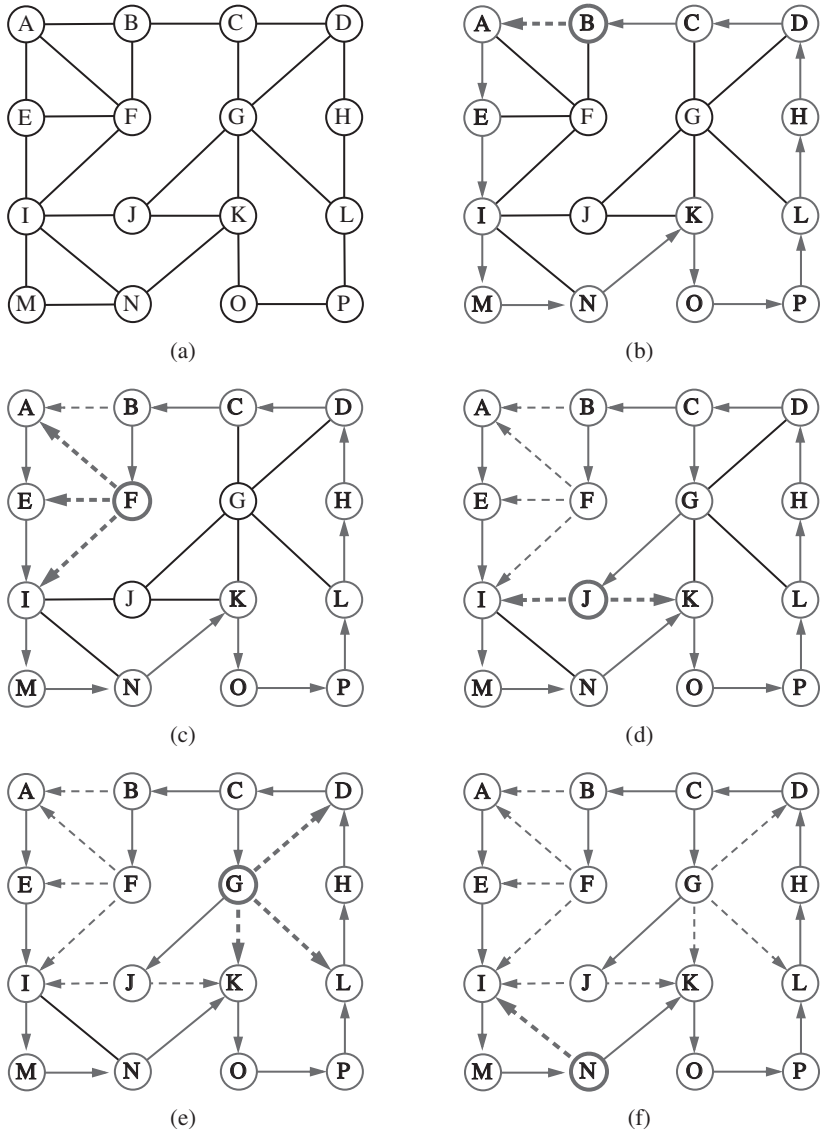


Figura 6.6 Exemplo de um caminhamento em profundidade em um grafo, iniciando no vértice A. Arestas de descoberta são mostradas em linhas sólidas, e arestas de retorno são mostradas em linhas pontilhadas. O vértice corrente é desenhado em linha mais grossa: (a) grafo de entrada; (b) caminho de arestas de descoberta a partir de A até que o retorno (B, A) é atingido; (c) alcançamos F, que é um local sem saída; (d) após retornar a C, continuamos pela aresta (C, G) e chegamos a outro local sem saída; (e) após retornar a G; (f) após retornar a N.

O Teorema 6.12 a seguir identifica outras propriedades importantes do método de caminhamento em profundidade.

Algoritmo DFS (G,v):
  Entrada: um grafo G e um vértice v de G.
  Saída: as arestas do componente conexo de v rotuladas como "descoberta" ou "retorno".
    rotule v como explorada
  para todos vértices e em G .incidentEdges(v) **faça**
    se aresta e é inexplorada **então**
      $w \leftarrow G$.opposite(v, e)
      se vértice w é inexplorado **então**
        rotule e como sendo de "descoberta"
        chame DFS(G,w)
      senão
        rotule e como sendo de "retorno"

Algoritmo 6.7 O algoritmo recursivo de caminhamento em profundidade.

Teorema 6.12: *seja G um grafo não-dirigido no qual um caminhamento em profundidade iniciando no vértice s foi realizado. Então o caminhamento visita todos os vértices no componente conexo de s e as arestas de descoberta formam uma árvore de cobertura do componente conexo de s.*

Prova: a prova é baseada em um argumento por contradição. Suponha que há pelo menos um vértice v não visitado no componente conexo de s, e seja w o primeiro vértice não visitado em um caminho de s a v (naturalmente, podemos ter $v = w$). Já que w é o primeiro vértice não visitado neste caminho, ele tem um vizinho u que foi visitado. Mas quando visitamos u, devemos ter considerado a aresta (u, w); portanto, não pode ser correto o fato de que w não foi visitado. Então, não há vértices não visitados no componente conexo de s. Já que somente marcamos arestas quando vamos a vértices não visitados, nunca formamos um ciclo com vértices de descoberta, ou seja, vértices de descoberta formam uma árvore. Além disso, ela é uma árvore de cobertura porque, como vimos, o caminhamento em profundidade visita cada vértice no componente conexo de s. ■

Note que o DFS é chamado uma vez a cada vértice, e que cada aresta é examinada exatamente duas vezes – uma vez em cada um de seus pontos finais. Assim, se m_s arestas estão no componente conexo do vértice s, um caminhamento em profundidade iniciado em s é executado em tempo $O(m_s)$, contanto que as condições a seguir sejam satisfeitas:

- O grafo é representado por uma estrutura de dados com os seguintes limites de desempenho:
  - o método incidentEdges(v) custa tempo O(degree(v));
  - os métodos hasNext( ) e nextEdge( ) do EdgeIterator retornado por incidentEdges(v) custam tempo $O(1)$ cada;
  - o método opposite(v, e) custa tempo $O(1)$.

A lista de adjacência satisfaz as propriedades acima, mas a matriz de adjacência não satisfaz.

- Temos uma maneira de "marcar" um vértice ou aresta à medida que exploramos e de testar se um vértice ou aresta foi explorado em tempo $O(1)$. Uma maneira de fazer essa marcação é estender a funcionalidade das posições que implementam vértices e arestas, adicionando um booleano visited. Outra maneira é usar o padrão de projeto decorador, discutido na Seção 6.5.

Teorema 6.13: *seja G um grafo com n vértices e m arestas representado com uma lista de adjacência. Um caminhamento em profundidade de G pode ser realizado em tempo O(n + m). Além disso, existem algoritmos de tempo O(n + m) baseados no caminhamento em profundidade para os seguintes problemas:*

- *testar se G é conexo;*
- *computar uma floresta de cobertura de G;*
- *computar os componentes conexos de G;*
- *computar um caminho entre dois vértices de G, ou reportar que um tal caminho não existe;*
- *computar um ciclo em G, ou reportar que G não tem ciclos.*

Exploraremos os detalhes da prova desse teorema em vários exercícios.

6.3.2 Componentes biconexos

Seja G um grafo conexo. Uma ***aresta de separação*** de G é uma aresta cuja remoção desconecta G. Um ***vértice de separação*** é um vértice cuja remoção desconecta G. Arestas e vértices de separação correspondem a pontos de falha em uma rede; portanto; freqüentemente desejamos identificá-los. Um grafo conexo G é ***biconexo*** se para quaisquer dois vértices u e v de G existirem dois caminhos disjuntos entre u e v, ou seja, dois caminhos que não compartilham arestas ou vértices comuns exceto u e v. Um ***componente biconexo*** de G é um subgrafo que satisfaz uma das seguintes condições (veja a Figura 6.8):

- um subgrafo de G que é biconexo e para o qual adicionar vértices ou arestas adicionais de G faria com que ele deixasse de ser biconexo;
- uma única aresta de G consistindo em uma aresta de separação e seus nodos inicial e final.

Se G for biconexo, ele tem um componente biconexo: o próprio G. Se G não tiver ciclos, por outro lado, então cada aresta de G é um componente biconexo. Componentes biconexos são importantes em redes de computadores, onde vértices representam roteadores e arestas representam conexões, pois, mesmo se um roteador em um componente biconexo falhar, mensagens ainda podem ser roteadas naquele componente usando os roteadores restantes.

Como afirmado no lema a seguir, cuja prova é deixada como exercício (C-6.5), a biconectividade é equivalente à ausência de vértices e arestas de separação.

Lema 6.14: *seja G um grafo conexo. As seguintes afirmativas são equivalentes:*

1. *G é biconexo.*
2. *Para quaisquer dois vértices de G, há um ciclo simples que os contém.*
3. *G não possui vértices ou arestas de separação.*

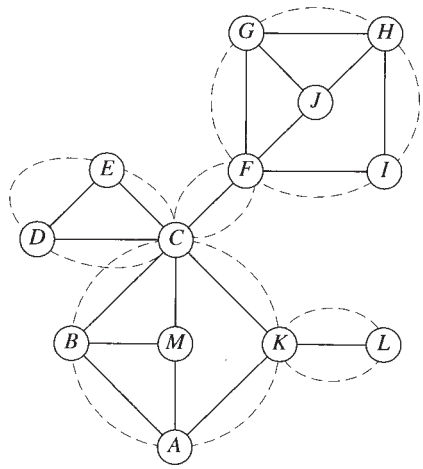


Figura 6.8 Componentes biconexos, mostrados circulados com linhas pontilhadas. C, F e K são vértices de separação; (C, F) e (K, L) são arestas de separação.

Classes de equivalência e a relação de ligação

Sempre que tivermos uma coleção C de objetos, podemos definir uma relação booleana $R(x,y)$ para cada par x e y em C. Ou seja, $R(x,y)$ é definida para cada x e y em C como sendo verdadeira ou falsa. A relação R é uma ***relação de equivalência*** se tiver as seguintes propriedades:

- ***Reflexividade:*** $R(x, x)$ é verdadeira para todo x em C.
- ***Simetria:*** $R(x,y) = R(y, x)$ é verdadeira para todo par x e y em C.
- ***Transitividade:*** se $R(x,y)$ for verdadeira e $R(y,z)$ for verdadeira, então $R(x,z)$ é verdadeira para todo x, y e z em C.

Por exemplo, o operador de igualdade (=) forma uma relação de equivalência em qualquer conjunto de números. A ***classe de equivalência*** para qualquer objeto x em C é o conjunto de todos os objetos y tais que $R(x,y)$ é verdadeira. Note que qualquer relação de equivalência R em um conjunto C divide o conjunto C em subconjuntos disjuntos que consistem nas classes de equivalência para os objetos em C.

Podemos definir uma interessante ***relação de ligação*** entre as arestas de um grafo G. Dizemos que duas arestas e e f de G são ***ligadas*** se $e = f$ ou se G tiver um ciclo simples contendo e e f. O lema a seguir apresenta propriedades fundamentais da relação de ligação:

Lema 6.15: *seja G um grafo conexo. Então,*

1. *A relação de ligação forma uma relação de equivalência para as arestas de G.*
2. *Um componente biconexo de G é o subgrafo induzido por uma classe de equivalência de arestas ligadas.*
3. *Uma aresta e de G é uma aresta de separação se e somente se e formar uma classe de equivalência de arestas ligadas com apenas um elemento.*
4. *Um vértice v de G é um vértice de separação se e somente se v tiver arestas incidentes em ao menos duas classes de equivalência distintas de arestas ligadas.*

Prova: é fácil ver que a relação de ligação é reflexiva e simétrica. Para mostrar que ela é transitiva, suponha que as arestas f e i são ligadas e as arestas g e h são ligadas. Se $f = g$ ou $g = h$, então $f = h$ ou existe um ciclo simples contendo f e h; portanto, f e h são ligadas. Suponha que f, g e h são distintas, ou seja, existe um ciclo simples C_{fg} passando por f e g e existe um ciclo simples C_{gh} passando por g e h. Considere o grafo obtido pela união dos ciclos C_{fg} e C_{gh}. Enquanto esse grafo pode não ser propriamente um ciclo simples, (embora pudéssemos ter $C_{fg} = C_{gh}$), ele contém um ciclo simples C_{fh} passando por f e h. Assim, f e h são ligadas e, portanto, a relação de ligação é uma relação de equivalência.

A correspondência entre classes de equivalência da relação de ligação e os componentes biconexos de G é uma conseqüência do Lema 6.14. ■

Uma abordagem de ligações para determinar componentes biconexos com DFS

Já que as classes de equivalência da relação de ligação sobre as arestas de G são as mesmas que os componentes biconexos de G, pelo Lema 6.15 podemos construir os componentes biconexos de G determinando as classes de equivalência da relação de ligação entre as arestas de G. Para isso, começamos com um caminhamento em profundidade de G, e construímos um ***grafo auxiliar*** B como segue (veja a Figura 6.9):

- Os vértices de B são as arestas de G.
- Para toda aresta de retorno e de G, sejam $f_1,...,f_k$ as arestas de descoberta de G que formam um ciclo com e. O grafo B contém as arestas $(e,f_1),...,(e,f_k)$.

Como existem $m - n + 1$ arestas de retorno e cada ciclo induzido por uma aresta de retorno tem no máximo $O(n)$ arestas, o grafo B contém no máximo $O(mn)$ arestas.

Um algoritmo de tempo $O(mn)$

Pela Figura 6.9, é aparente que cada componente conexo de B corresponde a uma classe de equivalência na relação de ligação para o grafo G. Depois disso, incluímos uma aresta (e,f) em B para cada aresta de retorno e encontrada no ciclo contendo f que foi induzida por e e pela árvore de cobertura mínima do caminhamento em profundidade.

O lema a seguir, cuja prova é deixada como exercício (C-6.7), estabelece uma forte relação entre o grafo B e as classes de equivalência na relação de ligação nos componentes de G, onde por brevidade chamamos as classes de equivalência da relação de ligação de ***componentes ligados*** de G.

Lema 6.16: *os componentes conexos do grafo auxiliar B correspondem aos componentes ligados do grafo G que induziram B.*

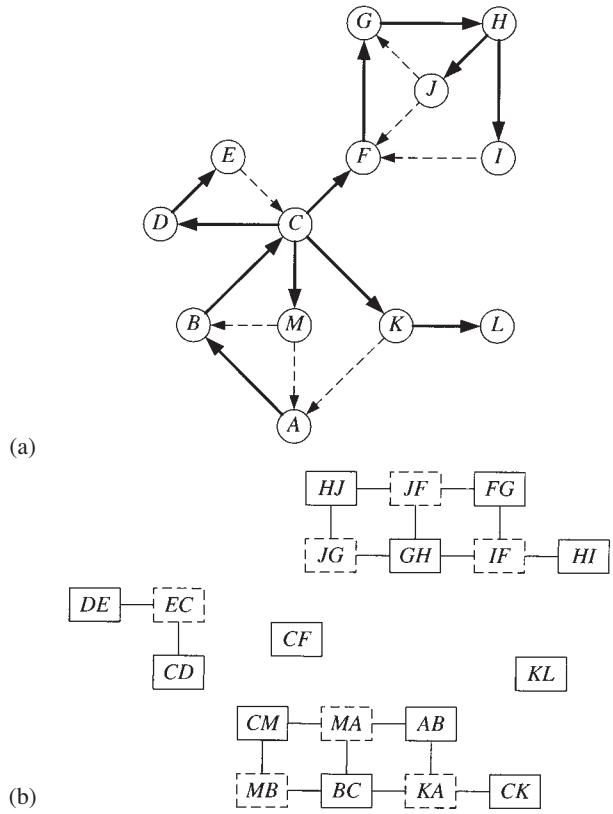


Figura 6.9 O grafo auxiliar usado para calcular componentes ligados: (a) o grafo G no qual o caminhamento em profundidade foi realizado (as arestas de retorno são traçadas em pontilhado); (b) o grafo auxiliar associado a G e seu caminhamento em profundidade.

O Lema 6.16 fornece o algoritmo de tempo $O(mn)$ a seguir para calcular todos os componentes ligados do grafo G com n vértices e m arestas:

1. realizar um caminhamento em profundidade T em G;
2. determinar o grafo auxiliar B identificando os ciclos de G induzidos por cada aresta de retorno de T;
3. determinar os componentes conexos de B, por exemplo, realizando um caminhamento em profundidade sobre B;
4. para cada componente conexo de B, apresentar os vértices de B (que são arestas de G) como um componente ligado de G.

Pela identificação dos componentes ligados de G, podemos determinar seus componentes biconexos, vértices de separação e arestas de separação do grafo G em tempo linear. Depois que as arestas de G foram divididas em classes de equivalência com respeito à relação de ligação, os componentes biconexos, vértices de separação e arestas de separação podem ser identificados em tempo $O(n + m)$ usando-se as regras simples listadas no Lema 6.15. Infelizmente, construir o grafo auxiliar B pode custar tempo $O(mn)$; portanto, o gargalo de todo o processo nesse algoritmo é a sua construção.

Note que não precisamos de todo o grafo auxiliar B para achar os componentes biconexos de G. Somente precisamos identificar os componentes conexos em B. Assim, seria suficiente se determinássemos uma árvore de cobertura para cada componente conexo em B, ou seja, uma floresta de cobertura de B. Como os componentes conexos de uma floresta de cobertura de B são iguais aos de B, não precisamos de todas as arestas de B – apenas suficientes delas para construir uma floresta de cobertura de B.

Portanto, vamos nos concentrar em como aplicar esta abordagem mais eficiente baseada em florestas de cobertura para determinar classes de equivalência para as arestas de G com respeito à relação de ligação.

Um algoritmo de tempo linear

Como descrito acima, podemos reduzir o tempo gasto para determinar os componentes ligados de G para o tempo $O(m)$ usando um grafo auxiliar de tamanho menor, que é uma floresta de cobertura de B. O algoritmo é descrito no Algoritmo 6.10.

Algoritmo LinkComponents(G):
  Entrada: um grafo conexo G
  Saída: os componentes ligados de G
Seja F um grafo auxiliar inicialmente vazio.
Realize um caminhamento em profundidade de G iniciando em
  um vértice arbitrário s.
Adicione cada vértice de descoberta f como um vértice em F
  e marque f como "não-ligado".
Para cada vértice v de G, seja $p(v)$ o pai de v na árvore de cobertura
  produzida pelo caminhamento em profundidade.
para cada vértice v em ordem crescente de visitação no caminhamento **faça**
  para cada aresta de retorno $e = (u,v)$ com destino v **faça**
    Adicione e como um vértice do grafo F.
    {Caminhe de u para s adicionando arestas a F se for necessário}
    enquanto $u \neq v$ **faça**
      Seja f o vértice em F correspondente à aresta de descoberta $(u,p(u))$.
      Adicione a aresta (e,f) a F.
      se f está marcado como "não-ligado" **então**
        Marque f como "ligado"
        $u \leftarrow p(u)$
      senão
        $u \leftarrow v$ {atalho para o fim do laço enquanto}
  Determine os componentes conexos do grafo F.

Algoritmo 6.10 Um algoritmo de tempo linear para determinar os componentes ligados. Note que um componente conexo de F que consiste em um vértice individual e "não-ligado" corresponde a uma aresta de separação (relacionada apenas consigo mesma).

Vamos analisar o tempo de execução de LinkComponents no Algoritmo 6.10. O caminhamento em profundidade inicial de G leva tempo $O(m)$. O maior trabalho, no entanto, é a construção do grafo auxiliar F, que leva tempo proporcional ao número de vértices e arestas em F. Note que, em algum ponto da execução do algoritmo, cada aresta de G é adicionada como vértice em F. Usamos um método de contabilização para as arestas de F: cada vez que adicionamos a F uma aresta (e, f), de uma aresta de retorno recém-encontrada e a uma aresta de descoberta f, vamos cobrar de f por esta ope-

ração se f estiver marcado como "não-ligado"; senão, cobraremos de e. Pela construção do laço **enquanto**, vemos que cobramos de cada vértice de F no máximo uma vez durante o algoritmo. Concluímos que a construção de F custa tempo $O(m)$. Finalmente, a determinação dos componentes conexos de F, que correspondem aos componentes ligados de G, custa tempo $O(m)$.

A correção do algoritmo anterior deriva do fato de que o grafo F em LinkComponents é uma floresta de cobertura do grafo B mencionado no Lema 6.16. Para detalhes, veja o Exercício C-6.8. Resumimos com o teorema a seguir e damos um exemplo de LinkComponents na Figura 6.11.

Teorema 6.17: *dado um grafo conexo G com m arestas, podemos determinar os componentes biconexos vértices de separação e arestas de separação de G, em tempo O(m).*

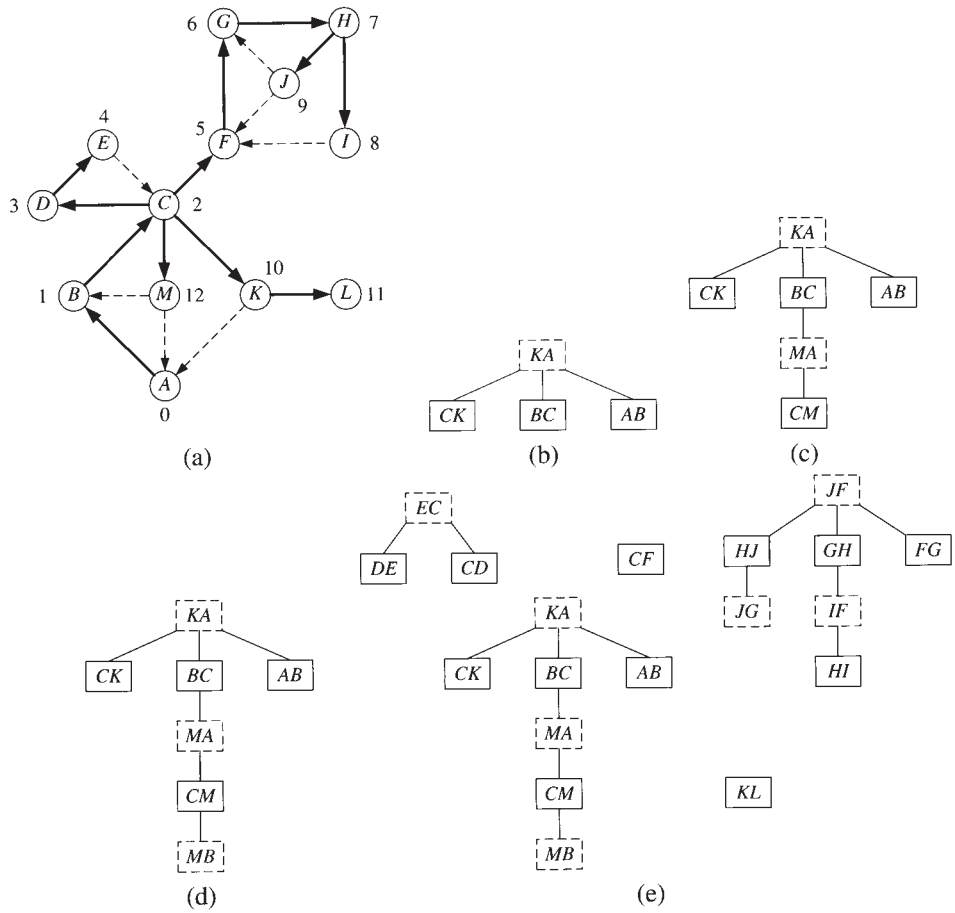


Figura 6.11 Exemplo da execução do algoritmo LinkComponents (Algoritmo 6.10): (a) grafo de entrada G depois do caminhamento em profundidade (os vértices estão rotulados por sua ordem de visitação e as arestas de retorno estão marcadas em pontilhado); grafo auxiliar F após processar (b) aresta de retorno (K, A), (c) aresta de retorno (M, A) e (d) aresta de retorno (M, B); (e) grafo F ao final do algoritmo.

6.3.3 Caminhamento em largura

Nesta seção, consideraremos o ***caminhamento em largura*** (BFS – *breadth-first search*). Como o caminhamento em profundidade, o caminhamento em largura percorre o componente conexo de um grafo e, ao fazê-lo, define uma árvore de cobertura bastante útil. O caminhamento em largura é menos "aventureiro" do que o caminhamento em profundidade, no entanto. Em vez de peregrinar pelo grafo, o caminhamento em largura atua em etapas e divide os vértices em ***níveis***. O caminhamento em largura também pode ser considerado como um caminhamento que usa um fio e tinta, desenrolando o fio de uma forma mais conservadora.

O caminhamento em largura inicia em um vértice s, que está no nível 0 e define a "âncora" para o fio. Na primeira etapa, desenrolamos o fio no comprimento de uma aresta e visitamos todos os vértices que podemos alcançar sem desenrolar mais fio. Neste caso, visitamos e pintamos como "visitados" os vértices adjacentes ao vértice inicial s – esses vértices são colocados no nível 1. Na segunda etapa, desenrolamos o fio no comprimento de duas arestas e visitamos todos os novos vértices que podemos alcançar sem desenrolar mais fio. Esses novos vértices, que são adjacentes aos vértices do nível 1 e não foram associados anteriormente a um nível, são colocados no nível 2, e assim por diante. O caminhamento em largura termina quando todos os vértices foram visitados.

O pseudocódigo para um caminhamento em largura iniciando em um vértice s é mostrado no Algoritmo 6.12. Usamos espaço auxiliar para rotular arestas, marcar vértices visitados e guardar contêineres associados com os níveis. Ou seja, os contêineres L_0, L_1, L_2 e assim por diante armazenam os vértices que estão no nível 0, nível 1, nível 2 e assim por diante. Estes contêineres poderiam, por exemplo, ser implementados como filas. Eles também permitem que o caminhamento em largura não seja recursivo.

```
Algoritmo BFS (G, s):
    Entrada: um grafo G e um vértice s de G
    Saída: as arestas do componente conexo de s rotuladas como
      "descoberta" ou "retorno"
    inicialize contêiner L_0
    para conter o vértice s
    i ← 0
    enquanto Li não está vazia faça
      crie o contêiner L_{i+1}, inicializando-o vazio
      para cada vértice v em L_i faça
        para todas as arestas e em G.incidentEdges(v) faça
          se aresta e é inexplorada então
            seja w o outro ponto final de e
            se o vértice w é inexplorado então
                rotule e como um aresta de descoberta
                insira w em L_{i+1}
            senão
                rotule e como uma aresta de cruzamento
      i ← i + 1
```

Algoritmo 6.12 Caminhamento em largura de um grafo G.

Ilustramos um caminhamento em largura na Figura 6.13.

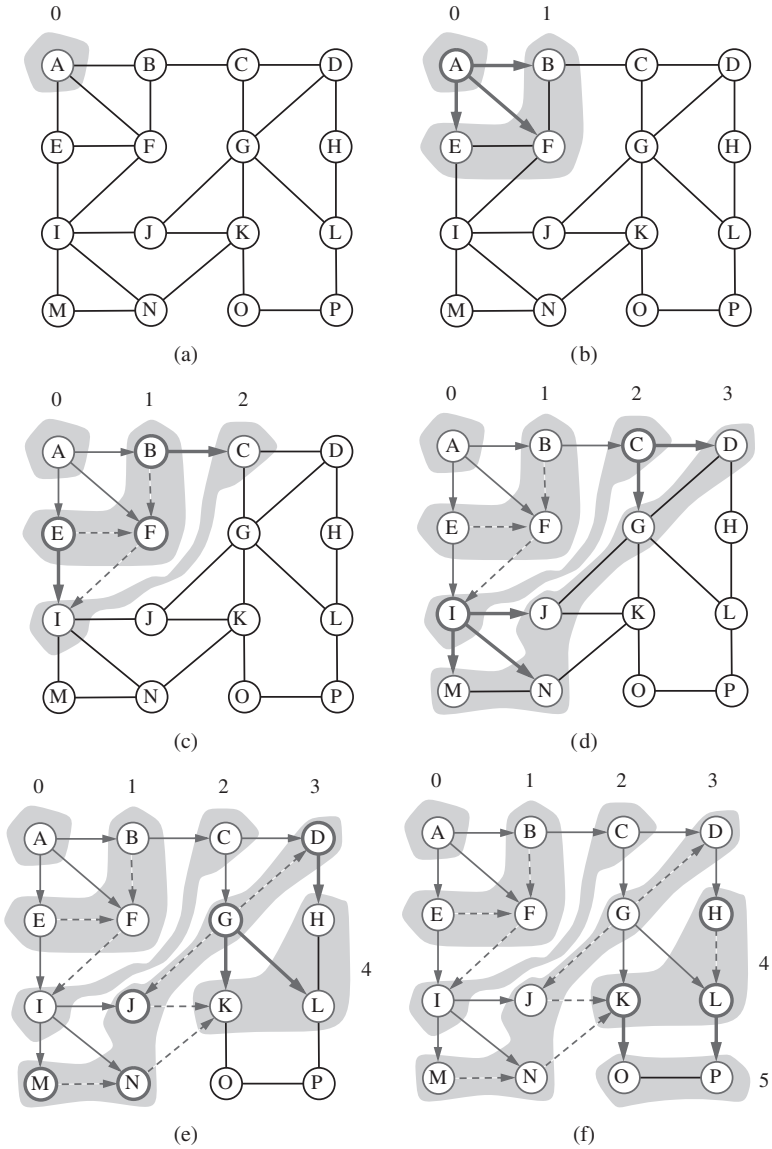


Figura 6.13 Exemplo de caminhamento em largura em que as arestas incidentes a um vértice são exploradas na ordem alfabética dos vértices adjacentes. As arestas de descoberta são mostradas com linhas sólidas e as arestas de cruzamento são mostradas com linhas pontilhadas: (a) o grafo antes do caminhamento; (b) descoberta do nível 1; (c) descoberta do nível 2; (d) descoberta do nível 3; (e) descoberta do nível 4; (f) descoberta do nível 5.

Uma das boas propriedades do caminhamento em largura é que durante o caminhamento podemos rotular cada vértice com o comprimento do menor caminho (em termos de número de arestas) desde o vértice inicial s. Em particular, se o vértice v for

colocado no nível i pelo caminhamento iniciado em s, então o comprimento do menor caminho de s a v é i.

Assim como o caminhamento em profundidade, podemos visualizar o caminhamento em largura orientando as arestas de acordo com a direção pela qual são exploradas durante o caminhamento e distinguindo as arestas usadas para descobrir novos vértices, chamadas de ***arestas de descoberta***, e aquelas que levam a vértices já visitados, ou ***arestas de cruzamento*** (veja a Figura 6.13f). Assim como o caminhamento em profundidade, as arestas de descoberta formam uma árvore de cobertura, que neste caso chamamos de árvore BFS. Não chamamos as arestas fora da árvore de "arestas de retorno" neste caso, no entanto, pois nenhuma delas conecta um vértice a um de seus antecedentes. Cada aresta fora da árvore conecta um vértice v a outro vértice que não é nem ancestral nem descendente de v.

O algoritmo de caminhamento em largura tem várias propriedades interessantes, algumas das quais são exploradas no teorema que segue.

Teorema 6.18: *seja G um grafo não-dirigido no qual um caminhamento em largura iniciado em um vértice s foi realizado. Então,*

- *o caminhamento visita todos os vértices no componente conexo de s;*
- *as arestas de descoberta formam uma árvore de cobertura T que chamamos de árvore de cobertura BFS para o componente conexo de s;*
- *para cada vértice v no nível i, o caminho na árvore T de s a v tem i arestas, e qualquer outro caminho em G de s a v tem pelo menos i arestas;*
- *se (u,v) for uma aresta de cruzamento, então os números dos níveis de u e v diferem em no máximo* 1.

Deixamos a justificativa dessa proposição como exercício (C-6.20). A análise do tempo de execução do caminhamento em largura é similar à do caminhamento em profundidade.

Teorema 6.19: seja G um grafo com n vértices e m arestas representado com uma lista de adjacência. Um caminhamento em largura de G leva tempo $O(n + m)$. Também existem algoritmos de tempo $O(n + m)$ baseados no caminhamento em largura para os seguintes problemas:

- testar se G é conexo;
- computar uma floresta de cobertura de G;
- computar os componentes conexos de G;
- dado um vértice inicial s de G, computar para cada vértice v de G um caminho com o menor número de vértices entre s e v ou reportar que não existe um caminho;
- computar um ciclo em G ou reportar que G não tem ciclos.

Comparando profundidade e largura

Pelo Teorema 6.19, o caminhamento em largura pode fazer tudo o que o caminhamento em profundidade faz. Há algumas diferenças interessantes entre esses dois métodos, no entanto, e existem algumas tarefas que um pode fazer melhor do que o outro. O caminhamento em largura é melhor para encontrar os caminhos mínimos em um grafo

(em que a distância é medida em número de arestas). Ele também produz uma árvore de cobertura tal que todas as arestas fora da árvore são arestas de cruzamento. O caminhamento em profundidade é melhor para responder perguntas complexas sobre conectividade, tais como determinar se cada par de vértices em um grafo pode ser conectado por dois caminhos disjuntos. Ele também produz uma árvore de cobertura tal que todas as arestas fora da árvore são arestas de retorno. Essas propriedades são válidas somente para grafos não-dirigidos, no entanto. Mesmo assim, como exploraremos na próxima seção, existem várias propriedades interessantes para as versões dirigidas do caminhamento em largura e do caminhamento em profundidade.

6.4 Grafos dirigidos

Nesta seção, consideraremos tópicos que são específicos de grafos dirigidos. Lembre que um grafo dirigido, também conhecido como ***dígrafo***, é um grafo em que todas as arestas são dirigidas.

Atingibilidade

Um dos tópicos mais fundamentais em grafos dirigidos é a noção de ***atingibilidade***, que trata de determinar onde podemos chegar em um grafo dirigido. Por exemplo, em uma rede de computadores com conexões unidirecionais, é importante saber se podemos atingir qualquer nodo a partir de qualquer outro nodo da rede. Um caminhamento em um grafo dirigido sempre segue a direção das arestas. Dados os vértices u e v de um dígrafo $\vec{G}$, dizemos que u ***atinge*** v (e v é ***atingido*** desde u) se $\vec{G}$ tem um caminho dirigido de u a v. Também dizemos que um vértice v atinge uma aresta (w, z) se v atinge o vértice w de origem da aresta.

Um dígrafo $\vec{G}$ é ***fortemente conexo*** se para quaisquer dois vértices u e v de $\vec{G}$ temos que u atinge v e v atinge u. Um ***ciclo dirigido*** de $\vec{G}$ é um ciclo onde todas as arestas são percorridas de acordo com suas direções respectivas. (Note que $\vec{G}$ pode ter um ciclo consistindo em duas arestas com direções opostas entre um par de vértices.) Um dígrafo $\vec{G}$ é ***acíclico*** se não tiver ciclos dirigidos (veja a Figura 6.14 para alguns exemplos).

O ***fechamento transitivo*** de um dígrafo $\vec{G}$ é o dígrafo $\vec{G}^*$, tal que os vértices de $\vec{G}^*$ são os mesmos de $\vec{G}$, e $\vec{G}^*$ tem um vértice (u,v) sempre que $\vec{G}$ tiver um caminho dirigido entre u e v. Ou seja, definimos $\vec{G}^*$ começando com o dígrafo $\vec{G}$ e adicionando uma aresta extra (u,v) para cada u e v tais que v é atingível a partir de u (e não existe ainda uma aresta (u,v) em $\vec{G}$).

Problemas interessantes que lidam com a atingibilidade em um dígrafo $\vec{G}$ incluem os seguintes:

- dados os vértices u e v, determinar se u atinge v;
- achar todos os vértices de $\vec{G}$ que são atingíveis de um dado vértice s;
- determinar se $\vec{G}$ é fortemente conexo;
- determinar se $\vec{G}$ é acíclico;
- computar o fechamento transitivo $\vec{G}^*$ de $\vec{G}$.

No restante desta seção, exploraremos alguns algoritmos eficientes para resolver esses problemas.

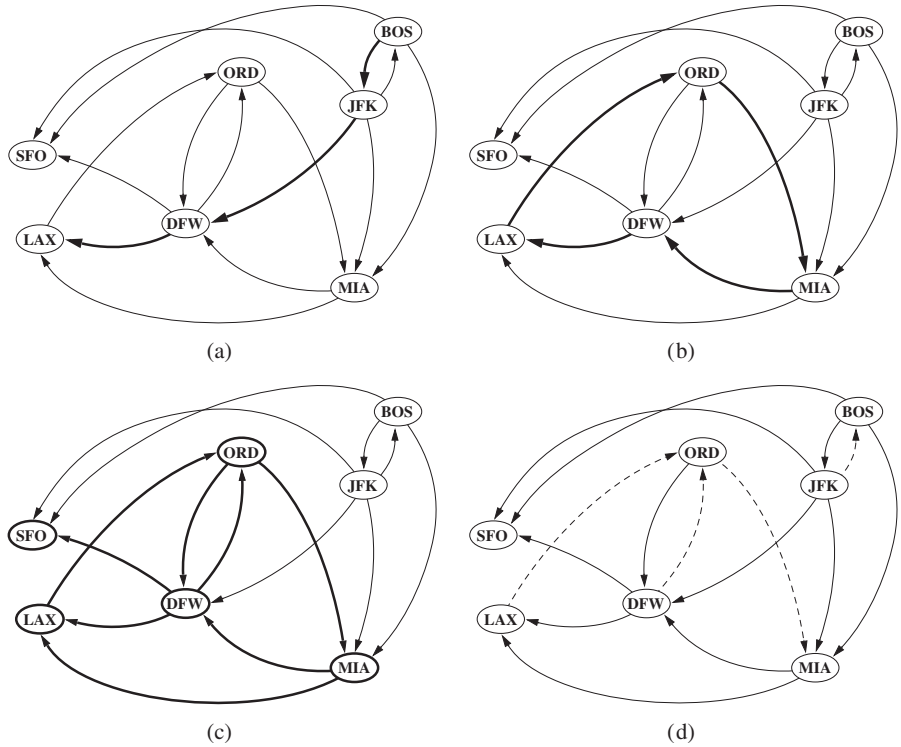


Figura 6.14 Exemplos de atingibilidade em um dígrafo: (a) um caminho dirigido de BOS a LAX é desenhado com linhas espessas; (b) um ciclo dirigido (ORD, MIA, DFW, LAX, ORD) é mostrado em linhas espessas; seus vértices formam um subgrafo fortemente conexo; (c) o subgrafo dos vértices e arestas atingíveis de ORD é mostrado com linhas espessas; (d) remover as arestas pontilhadas produz um dígrafo acíclico.

6.4.1 Caminhamento em um dígrafo

Assim como grafos não-dirigidos, podemos explorar um dígrafo de forma sistemática com métodos semelhantes ao caminhamento em largura (BFS) e ao caminhamento em profundidade (DFS) definidos previamente para grafos não-dirigidos (Seções 6.3.1 e 6.3.3). Estas explorações podem ser usadas, por exemplo, para resolver questões de atingibilidade. Os métodos dirigidos para caminhamento em profundidade e caminhamento em largura que desenvolveremos nesta seção para estas tarefas são muito semelhantes aos seus correspondentes não-dirigidos. De fato, a única diferença real é que esses métodos dirigidos para caminhamento em profundidade e caminhamento em largura somente percorrem as arestas de acordo com suas direções respectivas.

A versão dirigida do caminhamento em profundidade (DFS) iniciando no vértice v é ilustrada na Figura 6.15 (veja a Figura 6.9).

Um caminhamento em profundidade em um dígrafo $\vec{G}$ divide as arestas de $\vec{G}$ atingíveis pelo vértice inicial em ***arestas de árvore*** ou ***arestas de descoberta***, que nos levam a descobrir um novo vértice, e ***arestas fora da árvore***, que nos levam a um vérti-

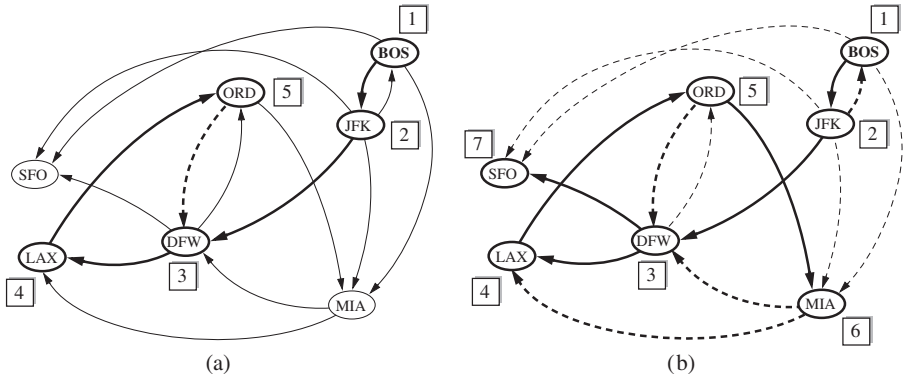


Figura 6.15 Um exemplo de caminhamento em profundidade em um dígrafo: (a) passo intermediário, em que pela primeira vez um vértice já visitado (DFW) é alcançado; (b) o caminhamento em profundidade completo. As arestas da árvore são mostradas com linhas espessas, as arestas de retorno são mostradas com linhas pontilhadas espessas e as arestas de descoberta e de cruzamento são mostradas com linhas pontilhadas finas. A ordem na qual os vértices são visitados é indicada pelo número próximo ao vértice. A aresta (ORD, DFW) é uma aresta de retorno, mas (DFW, ORD) é uma aresta de descoberta. A aresta (BOS, SFO) é uma aresta de descoberta e (SFO, LAX) é uma aresta de cruzamento.

ce visitado previamente. As arestas de árvore formam uma árvore com raiz no vértice inicial, chamada de ***árvore de profundidade***, e existem três tipos de arestas fora da árvore (veja a Figura 6.15b):

- ***arestas de retorno***, que conectam um vértice a seu antecessor na árvore de profundidade;
- ***arestas de descoberta***, que conectam um vértice a um descendente na árvore de profundidade;
- ***arestas de cruzamento***, que conectam um vértice a um vértice que não é nem seu antecessor nem seu descendente.

Teorema 6.20: *seja $\vec{G}$ um dígrafo. O caminhamento em profundidade em $\vec{G}$ iniciando em um vértice s visita todos os vértices de $\vec{G}$ que são atingíveis desde s. A árvore de profundidade contém caminhos dirigidos de s a qualquer vértice atingível a partir de s.*

Prova: seja V_s o subconjunto de vértices de $\vec{G}$ visitados pelo caminhamento em profundidade iniciando em um vértice s. Desejamos mostrar que V_s contém s e qualquer vértice atingível a partir de s. Suponha, para obter uma contradição, que haja um vértice w atingível desde s que não está em V_s. Considere um caminho dirigido de s a w e seja (u,v) a primeira aresta deste caminho que nos retira de V_s, ou seja, u está em V_s e v não está. Quando o caminhamento em profundidade atinge u, ele explora todas as arestas saindo de u e, por isso, deve atingir também o vértice v via a aresta (u,v). Portanto, v deve estar em V_s, e obtivemos uma contradição. Assim, V_s deve conter cada vértice atingível a partir de s. ■

Analisar o tempo de execução do caminhamento em profundidade dirigido é análogo ao seu equivalente não-dirigido. Uma chamada recursiva é feita para cada vértice

uma vez, e cada aresta é percorrida uma vez (desde sua origem). Assim, se o subgrafo atingível desde o vértice s tem m_s arestas e n_s vértices, um caminhamento em profundidade dirigido iniciando em s é executado em tempo $O(n_s + m_s)$, desde que o dígrafo seja representado com uma lista de adjacência.

De acordo com o Teorema 6.20, podemos usar o caminhamento em profundidade para encontrar todos os vértices atingíveis a partir de um dado vértice e, portanto, achar o fechamento transitivo de $\vec{G}$, ou seja, podemos fazer um caminhamento em profundidade iniciando de cada vértice v em $\vec{G}$ para verificar quais vértices w são atingíveis a partir de v, adicionando uma aresta (v,w) ao fechamento transitivo para cada um destes vértices w. De forma similar, se percorrermos repetidamente o dígrafo $\vec{G}$ com o caminhamento em profundidade, iniciando a cada vez em um vértice diferente, podemos testar facilmente se $\vec{G}$ é fortemente conexo. Ou seja, $\vec{G}$ é fortemente conexo se cada caminhamento em profundidade visitar todos os vértices de $\vec{G}$.

Teorema 6.21: *seja $\vec{G}$ um dígrafo com n vértices e m arestas. Os problemas a seguir podem ser solucionados por um algoritmo executado em tempo $O(n(n + m))$:*

- *determinar, para cada vértice v de $\vec{G}$, o subgrafo atingível a partir de v;*
- *testar se $\vec{G}$ é fortemente conexo;*
- *determinar o fechamento transitivo $\vec{G}^*$ de $\vec{G}$.*

Teste para conectividade forte

Podemos determinar se um grafo dirigido $\vec{G}$ é fortemente conexo muito mais depressa do que o teorema determina, usando apenas dois caminhamentos em profundidade.

Começamos realizando um caminhamento em $\vec{G}$ iniciando em um vértice arbitrário s. Se existir algum vértice de $\vec{G}$ que não é visitado por este caminhamento e não é atingível a partir de s, então o grafo não é fortemente conexo. Assim, se este primeiro caminhamento visita cada vértice de $\vec{G}$, então revertemos todas as arestas de $\vec{G}$ (usando o método reverseDirection) e realizamos outro caminhamento em profundidade iniciando em s neste grafo "revertido". Se cada vértice de $\vec{G}$ for visitado por este segundo caminhamento, então o grafo é fortemente conexo, pois cada um dos vértices visitados no caminhamento pode atingir s. Já que este algoritmo realiza apenas dois caminhamentos em profundidade sobre $\vec{G}$, ele é executado em tempo $O(n + m)$.

Caminhamento em largura dirigido

Assim como o caminhamento em profundidade, podemos estender o caminhamento em largura para grafos dirigidos. O algoritmo ainda visita os vértices nível a nível e divide o conjunto de arestas em ***arestas de árvore*** (ou ***arestas de descoberta***), que juntas formam uma árvore dirigida ***em largura*** com raiz no vértice inicial e ***arestas fora da árvore***. Diferentemente do caminhamento em profundidade dirigido, o caminhamento em largura dirigido produz apenas dois tipos de arestas fora da árvore:

- ***arestas de retorno***, que conectam um vértice a um de seus antecessores;
- ***arestas de cruzamento***, que conectam um vértice a outro vértice que não é seu antecessor ou sucessor.

Não existem arestas de descoberta, um fato que exploraremos em um exercício (C-6.14).

6.4.2 Fechamento transitivo

Nesta seção, exploraremos uma técnica alternativa para determinar o fechamento transitivo de um dígrafo. Ou seja, descreveremos um método direto para determinar todos os pares de vértices (v,w) em um grafo dirigido tais que w é atingível desde v. Esse tipo de informação é útil, por exemplo, em redes de computadores, pois permite que saibamos imediatamente se podemos rotear uma mensagem de um nodo v para um nodo w, ou se seria apropriado responder "você não pode chegar lá" a respeito desta mensagem.

Seja $\vec{G}$ um grafo dirigido com n vértices e m arestas. Determinamos o fechamento transitivo de $\vec{G}$ em uma série de rodadas. Inicializamos $\vec{G}_0 = \vec{G}$. Também numeramos de forma arbitrária os vértices de $\vec{G}$ como

$$v_1, v_2, ..., v_n.$$

Iniciamos as rodadas começando pela rodada 1. Na rodada k, construímos um dígrafo $\vec{G}_k$ iniciando com $\vec{G}_k = \vec{G}_{k-1}$ e adicionando a $\vec{G}_k$ a aresta dirigida (v_i,v_j) se o dígrafo $\vec{G}_{k-1}$ contiver as arestas (v_i,v_k) e (v_k,v_j). Desta forma, forçaremos a aplicação de uma regra simples, explicada no lema que se segue.

Lema 6.22: *para $i = 1,...,n$, o dígrafo $\vec{G}_k$ tem uma aresta (v_i,v_j) se e somente se o dígrafo $\vec{G}$ tiver um caminho dirigido de v_i a v_j cujos vértices intermediários (se existirem) pertencem ao conjunto $\{v_1,...,v_k\}$. Em particular, $\vec{G}_n$ é igual a $\vec{G}^*$, o fechamento transitivo de $\vec{G}$.*

Esse lema sugere um algoritmo simples de ***programação dinâmica*** (Seção 5.3) para determinar o fechamento transitivo de $\vec{G}$, que é conhecido como o ***algoritmo de Floyd-Warshall***. O pseudocódigo para este método é mostrado no Algoritmo 6.16.

É fácil analisar o tempo de execução do algoritmo de Floyd-Warshall. O laço principal é executado n vezes e o laço interno considera cada um dos $O(n^2)$ pares de vértices, realizando uma operação de tempo constante para cada par. Se usarmos uma estrutura de dados como a matriz de adjacência, que suporta os métodos areAdjacent e insertDirectedEdge em tempo $O(1)$, temos um tempo de execução total de $O(n^3)$.

Algoritmo FloydWarshall($\vec{G}$):
  Entrada: um dígrafo $\vec{G}$ com n vértices.
  Saída: O fechamento transitivo $\vec{G}^*$ de $\vec{G}$
  sejam os vértices $v_1,...,v_n$ uma numeração arbitrária dos vértices de $\vec{G}$
  $\vec{G}_0 \leftarrow \vec{G}$
  para $k \leftarrow 1$ até n **faça**
    $\vec{G}_k \leftarrow \vec{G}_{k-1}$
    para $i \leftarrow 1$ até n com $i \neq k$ **faca**
      para $j \leftarrow 1$ até n com $j \neq i,k$ **faça**
        se arestas (v_i,v_k) e (v_k,v_j) estão em $\vec{G}_{k-1}$ **então**
          se $\vec{G}_k$ não contém a aresta dirigida (v_i,v_j) **então**
            adicione aresta (v_i,v_j) a $\vec{G}_k$
  retorne $\vec{G}_n$

Algoritmo 6.16 Algoritmo de Floyd-Warshal. Esse algoritmo de programação dinâmica computa o fechamento transitivo $\vec{G}^*$ de G calculando uma série de dígrafos $\vec{G}_0, \vec{G}_1, ..., \vec{G}_n$, para $k = 1,..., n$.

Pela descrição e análise anterior, podemos obter imediatamente o teorema a seguir.

Teorema 6.23: *seja $\vec{G}$ um dígrafo com n vértices, representado em uma matriz de adjacência. Então o algoritmo de Floyd-Warshall determina o fechamento transitivo $\vec{G}^*$ de $\vec{G}$ em tempo $O(n^3)$.*

Desempenho do algoritmo de Floyd-Warshall

Comparamos agora o tempo de execução do algoritmo de Floyd-Warshall com o do algoritmo, mais complicado, do Teorema 6.21, que repetidamente realiza n caminhamentos em profundidade, iniciados a cada vértice.

Se o dígrafo for representado usando-se uma matriz de adjacência, então executar o caminhamento em profundidade uma vez sobre o grafo dirigido $\vec{G}$ custa tempo $O(n^2)$ (exploraremos o motivo em um exercício). Assim, executar o caminhamento em profundidade n vezes custa tempo $O(n^3)$, o que não é melhor do que uma execução do algoritmo de Floyd-Warshall.

Se o grafo é representado usando-se uma lista de adjacência, então efetuar o caminhamento em profundidade n vezes tomaria tempo $O(n(n + m))$ para determinar o fechamento transitivo. Mesmo assim, se o grafo for ***denso***, ou seja, se ele tiver $\Theta(n^2)$ arestas, então esta abordagem ainda custa tempo $O(n^3)$.

O único caso em que o algoritmo do Teorema 6.21 é melhor do que o algoritmo de Floyd-Warshall é quando o grafo não for denso e for representado usando-se uma lista de adjacência.

Exemplificamos a execução do algoritmo de Floyd-Warshall na Figura 6.17.

6.4.3 Caminhamento em profundidade e coleta de lixo

Em algumas linguagens como C e C++, o espaço de memória para objetos deve ser explicitamente alocado e liberado pelo programador. Essa tarefa de gerenciamento de memória é freqüentemente desconsiderada por programadores iniciantes e, quando feita de forma incorreta, pode ser fonte de erros de programação frustrantes até mesmo para programadores experientes. Assim, os projetistas de outras linguagens, como Java, passam a tarefa do gerenciamento de memória ao ambiente de execução. Um programador Java não tem de liberar explicitamente a memória ocupada por um objeto quando a vida deste se encerra. Em vez disso, o mecanismo de ***coleta de lixo*** libera a memória destes objetos.

Em Java a memória para a maior parte dos objetos é alocada no *heap* (que não deve ser confundido com a estrutura de dados de mesmo nome). Além disso, as *threads* sendo executadas (veja a Seção 2.1.2) armazenam o espaço corrente para suas variáveis instanciadas em suas respectivas pilhas de métodos (veja a Seção 2.1.1). Já que as variáveis instanciadas nas pilhas de métodos podem se referir a objetos no *heap*, todas as variáveis e objetos nas pilhas das *threads* sendo executadas são chamadas de ***objetos raiz***. Todos os objetos que podem ser alcançados seguindo-se referências a objetos que se iniciam em um objeto raiz são chamadas ***objetos vivos***. Os objetos vivos são os objetos ativos correntemente usados pelo programa sendo executado; esses objetos ***não*** devem ser desalocados. Por exemplo, um programa Java sendo executado pode armazenar em uma variável uma referência a uma seqüência S que é implementada com

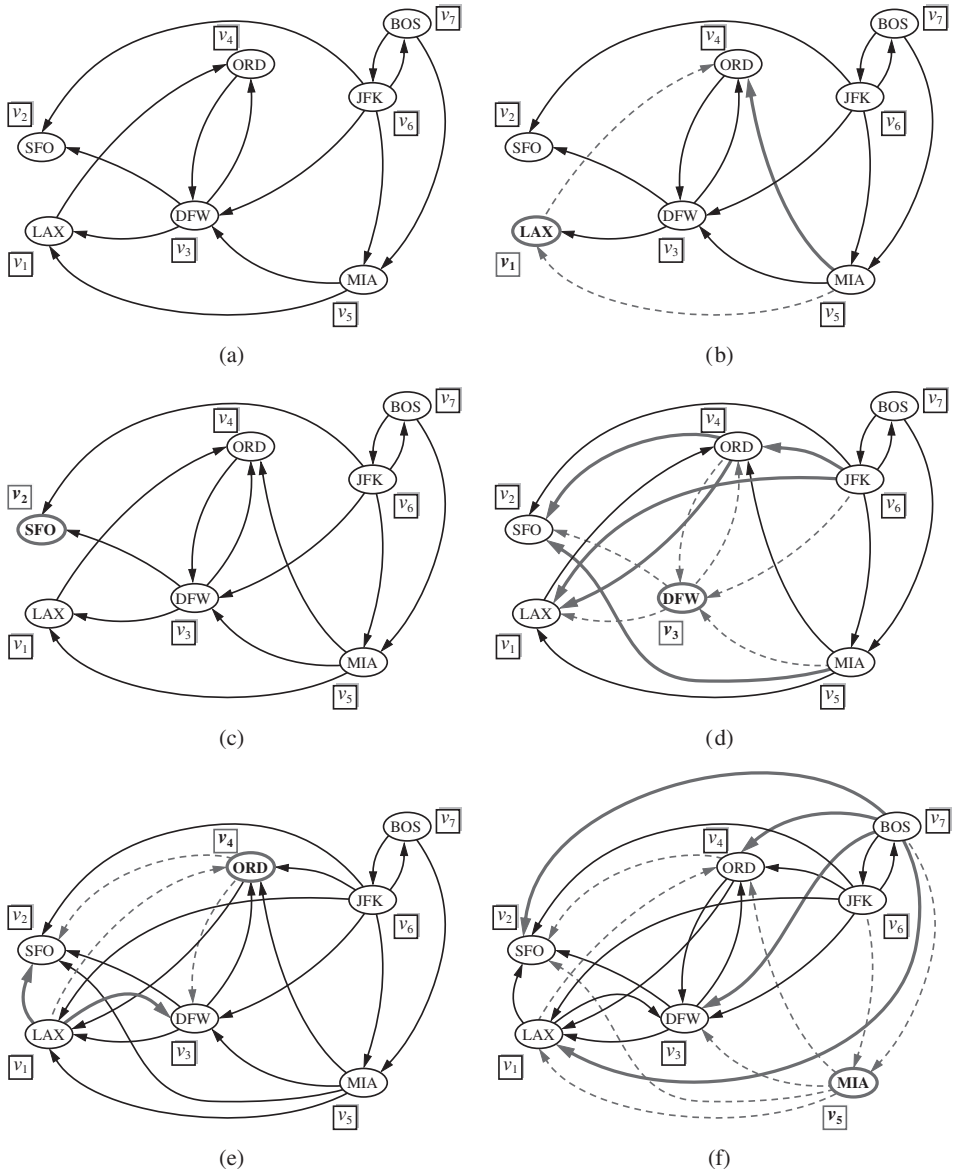


Figura 6.17 Seqüência de dígrafos determinados pelo algoritmo de Floyd-Warshall: (a) dígrafo inicial $\vec{G} = \vec{G}_0$ e enumeração dos vértices; (b) o dígrafo $\vec{G}_1$; (c)$\vec{G}_2$; (d)$\vec{G}_3$; (e)$\vec{G}_4$; (f)$\vec{G}_5$. Note que $\vec{G}_5 = \vec{G}_6 = \vec{G}_7$. Se o dígrafo $\vec{G}_{k-1}$ tem as arestas (v_i,v_k) e (v_k,v_j) mas não a aresta (v_i,v_j), no desenho do dígrafo $\vec{G}_k$ mostramos as arestas (v_i,v_k) e (v_k,v_j) com linhas pontilhadas e a aresta (v_i,v_j) com uma linha espessa.

uma lista duplamente encadeada. A variável que faz referência a *S* é um objeto raiz, enquanto o objeto para *S* é um objeto vivo, assim como são todos os nodos referenciados a partir deste objeto e todos os elementos referenciados por estes nodos.

De tempos em tempos, a máquina virtual Java (JVM) percebe que o espaço de memória livre no *heap* está ficando reduzido. Nestes momentos, a JVM provavelmente irá

requisitar o espaço sendo usado por objetos que não estão mais vivos. Esse processo de requisição é conhecido como ***coleta de lixo***. Existem vários algoritmos diferentes para coleta de lixo, e um dos mais usados é o algoritmo de ***marcação e varredura***.

O algoritmo de marcação e varredura

No algoritmo de coleta de lixo com marcação e varredura, associamos a cada objeto um *bit* de "marca" que diz se o objeto está vivo ou não. Quando determinamos em algum momento que a coleta é necessária, suspendemos todas as *threads* sendo executadas e limpamos os *bits* de marca dos objetos atualmente sendo armazenados no *heap*. Então percorremos as pilhas Java das *threads* sendo executadas e marcamos como "vivos" todos os objetos (raiz) destas pilhas. Devemos então determinar todos os outros objetos vivos – aqueles que são atingíveis a partir dos objetos raiz. Para fazer isso eficientemente, devemos usar a versão dirigida do algoritmo de caminhamento em profundidade. Neste caso, cada objeto no *heap* é visto como um vértice em um grafo dirigido, e a referência de um objeto a outro é vista como uma aresta. Realizando um caminhamento em profundidade dirigido a partir de cada objeto raiz, podemos identificar e marcar corretamente cada objeto vivo. Esse processo é conhecido como a fase de marcação. Quando ele é completado, percorremos o *heap* e recuperamos o espaço usado por qualquer objeto que não tenha sido marcado. Esse processo é chamado de varredura e quando ele é completado, continuamos a executar as *threads* que foram suspensas. Assim, o algoritmo de coleta de lixo com marcação e varredura busca o espaço não utilizado em tempo proporcional ao número de objetos vivos e suas referências mais o tamanho do *heap*.

Realização do caminhamento em profundidade no local

O algoritmo de marcação e varredura é uma forma efetiva de recuperar o espaço de memória não usado no *heap*, mas há um detalhe importante que deve ser encarado durante a fase de marcação. Já que estamos buscando obter mais memória quando esta é escassa, devemos tomar cuidado especial para não usar muito espaço adicional com a própria coleta de lixo. O problema é que o algoritmo de caminhamento em profundidade, como foi descrito, pode usar tanto espaço extra quantos forem os vértices do grafo. No caso da coleta de lixo, os vértices do grafo são os objetos do *heap*. Provavelmente não temos toda essa memória para usar; por isso, nossa única alternativa é encontrar uma forma de realizar o caminhamento em profundidade "no local", e não recursivamente, ou seja, devemos realizar o caminhamento em profundidade usando uma quantidade constante de memória auxiliar.

A idéia básica para fazer o caminhamento em profundidade no local é simular a pilha de recursão usando as arestas do grafo (que no caso da coleta de lixo correspondem a referências a objetos). Sempre que percorremos uma aresta de um vértice visitado v para um novo vértice w, alteramos a aresta (v, w) armazenada na lista de adjacência de v para apontar para o pai de v na árvore do caminhamento em profundidade. Quando retornamos a v (simulando o retorno da função "recursiva" em w), podemos alterar novamente a aresta para apontar de volta para w. É claro que precisamos ter uma forma de identificar qual a aresta que deve ser modificada. Uma possibilidade é numerar as referências saindo de v como 1, 2 e assim por diante e armazenar, além do *bit* de marcação (que estamos usando para guardar a informação de "visitado" no caminhamento), uma variável que nos informa que arestas foram modificadas.

Usar esta variável requer uma palavra extra de memória para cada objeto. Essa palavra extra, no entanto, pode ser evitada em algumas implementações. Por exemplo, muitas implementações da máquina virtual Java representam um objeto como a composição de uma referência com identificador de tipo (que indica se este objeto é um Integer ou Vector ou algum outro tipo) e uma referência a outros objetos ou dados deste objeto. Já que se supõe que as informações sobre tipos sejam os primeiros elementos da composição nestas implementações, podemos usar esta referência para "marcar" a aresta que alteramos quando saímos do objeto v e fomos para o objeto w no caminhamento. Simplesmente trocamos a referência em v que descreve o tipo de v com a referência em v que se refere a w. Quando retornamos a i, podemos identificar rapidamente a aresta (v, w) que alteramos, porque ela será a primeira referência na composição de v, e a posição da referência para o tipo de v nos informa a posição desta aresta na lista de adjacência de v. Assim, se usamos este truque de trocar arestas ou se usamos uma variável, podemos implementar o caminhamento em profundidade no local sem afetar seu tempo de execução assintótico.

6.4.4 Grafos acíclicos dirigidos

Grafos dirigidos sem ciclos dirigidos são encontrados em muitas aplicações. Esse tipo de dígrafo é freqüentemente chamado de ***grafo acíclico dirigido*** ou ***DAG*** (*directed acyclic graph*, em inglês). As aplicações desses grafos incluem as seguintes:

- herança entre classes em um programa Java;
- pré-requisitos entre disciplinas de um curso universitário;
- restrições de seqüenciamento entre tarefas de um projeto.

Exemplo 6.24: *para gerenciar um grande projeto, é conveniente dividi-lo em uma coleção de pequenas tarefas. As tarefas, no entanto, raramente são independentes, pois costumam existir restrições de seqüenciamento entre elas (por exemplo, na construção de uma casa, a tarefa de encomendar os pregos obviamente precede a tarefa de pregar as vigas do teto). É claro que as restrições de seqüenciamento não podem ter circularidades, pois elas tornariam o projeto impossível (por exemplo, para obter um emprego, você precisa ter experiência, mas para ter experiência você deve ter um emprego). As restrições limitam a ordem em que as tarefas podem ser executadas. Ou seja, se uma restrição diz que a tarefa a deve ser realizada antes da tarefa b ser iniciada, então a deve preceder b na execução das tarefas. Assim, se modelarmos o conjunto de tarefas como vértices de um grafo dirigido e colocarmos uma aresta dirigida de v para w sempre que a tarefa v tiver de ser executada antes da tarefa w, então estaremos definindo um grafo acíclico dirigido.*

O exemplo acima motiva a definição a seguir. Seja $\vec{G}$ um dígrafo com n vértices. Uma ***ordenação topológica*** de $\vec{G}$ é uma ordem $v_1,...,v_n$ dos vértices de $\vec{G}$ tal que para cada aresta (v_i,v_j) de $\vec{G}$, $i < j$. Ou seja, uma ordenação topológica é uma ordenação que faz com que um caminho dirigido em $\vec{G}$ passe pelos vértices de $\vec{G}$ em ordem crescente (veja a Figura 6.18). Note que um dígrafo pode ter mais de uma ordenação topológica.

Teorema 6.25: *$\vec{G}$ tem uma ordenação topológica se e somente se $\vec{G}$ for acíclico.*

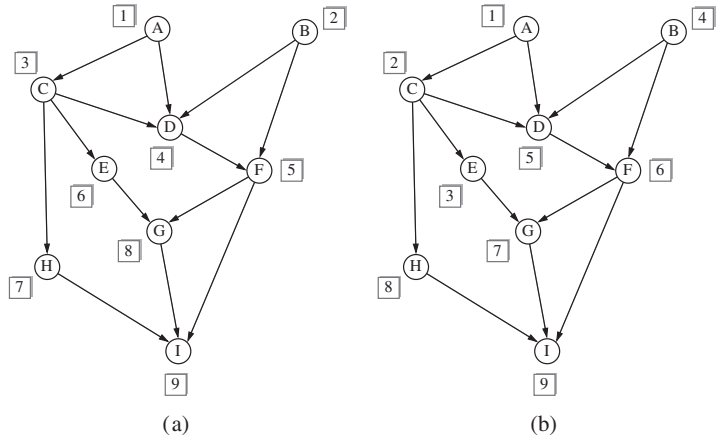


Figura 6.18 Duas ordenações topológicas do mesmo grafo acíclico dirigido.

Prova: a necessidade (o "somente se" da proposição) é fácil de demonstrar. Suponha que $\vec{G}$ seja topologicamente ordenado. Assuma, para obter uma contradição, que $\vec{G}$ tem um ciclo consistindo nas arestas (v_{i0}, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{ik-1}, v_{i0}). Por causa da ordenação topológica, devemos ter $i_0 < i_1 < \cdots < i_{k-1} < i_0$, o que é claramente impossível. Assim, $\vec{G}$ deve ser acíclico. Agora justificaremos a suficiência da condição (o "se" da proposição). Suponha que $\vec{G}$ for acíclico. Daremos uma descrição algorítmica de como construir uma ordenação topológica para $\vec{G}$. Como $\vec{G}$ é acíclico, $\vec{G}$ deve ter um vértice onde não chega nenhuma aresta (ou seja, um vértice com grau de entrada 0). Seja v_1 um vértice desse tipo. De fato, se v_1 não existir, então ao percorrer um caminho dirigido a partir de um vértice arbitrário, terminaríamos por encontrar um vértice visitado previamente, o que contradiz o fato de $\vec{G}$ ser acíclico. Se removermos v_1 de $\vec{G}$, bem como suas arestas de saída, o dígrafo resultante ainda é acíclico. Portanto, o dígrafo resultante também tem um vértice v_2 onde não chegam arestas. Repetindo esse processo até que o dígrafo esteja vazio, obtemos uma ordenação $v_1,...,v_n$ dos vértices de $\vec{G}$. Pela construção acima, se (v_i,v_j) é uma aresta de $\vec{G}$, então v_1 deve ser removido antes que v_j possa ser removido e, portanto, $i < j$. Assim, $v_1,...,v_n$ é uma ordenação topológica. ■

A prova acima sugere o Algoritmo 6.19, chamado de ***ordenação topológica***.

Teorema 6.26: *seja $\vec{G}$ um dígrafo com n vértices e m arestas. O algoritmo de ordenação topológica é executado em tempo $O(n + m)$ usando espaço auxiliar $O(n)$ e determina uma ordenação topológica de $\vec{G}$ ou não consegue numerar algum vértice, o que indica que $\vec{G}$ tem um ciclo dirigido.*

Prova: a determinação inicial dos graus de entrada e preparo das variáveis *incounter* podem ser feitos com uma passagem sobre o grafo, que custa tempo $O(n + m)$. Podemos adicionar um campo extra aos vértices do grafo ou usar o padrão de decoradores descrito na próxima seção para associar atributos a cada vértice. Um vértice u é ***visitado*** pelo algoritmo de ordenação topológica quando u é removido da pilha S. Um vértice u pode ser visitado somente quando $incounter(u) = 0$, o que implica que todos os seus predecessores (vértices com arestas que levam a u) foram visitados previamente. Como conseqüência, qualquer vértice que faz parte de um ciclo dirigido nunca será visitado e qualquer outro vértice será visitado exatamente uma vez. O algoritmo percorre todas as arestas saindo de cada vértice visitado exatamente uma vez; portanto, seu

Algoritmo TopologicalSort($\vec{G}$):
  Entrada: um dígrafo $\vec{G}$ com n vértices
  Saída: uma ordenação topológica $v_1,...,v_n$ de $\vec{G}$ ou uma indicação de que $\vec{G}$ tem um ciclo dirigido
  seja S uma pilha inicialmente vazia
  para cada vértice u de G **faça**
    incounter(u) ← indeg(u)
    se incounter(u) = 0 **então**
      S.push(u)
  $i \leftarrow 1$
  enquanto S não está vazia **faça**
    $u \leftarrow S$.pop( )
    numere u como o i-ésimo vértice v_i
    $i \leftarrow i + 1$
    para cada aresta $e \in \vec{G}$.outIncidentEdges(u) **faça**
      $w \leftarrow G$.opposite(u,e)
      incounter(w) ← incounter(w) –1
      se incounter(w) = 0 **então**
        S.push(w)
  se $i > n$ **então**
    retorne v_i , ... , v_n
  senão
    retorne "o dígrafo $\vec{G}$ tem um ciclo dirigido"

Algoritmo 6.19 Algoritmo de ordenação topológica.

tempo de execução é proporcional ao número de arestas saindo dos vértices visitados. Assim, o algoritmo é executado em tempo $O(n + m)$. Quanto ao espaço usado, observe que a pilha S e as variáveis *incounter* associadas aos vértices usam espaço $O(n)$.

Como um efeito secundário, o algoritmo também testa se um grafo $\vec{G}$ é acíclico. De fato, se o algoritmo termina sem ordenar todos os vértices, então o subgrafo dos vértices que não foram ordenados deve conter um ciclo dirigido (veja Figura 6.20). ■

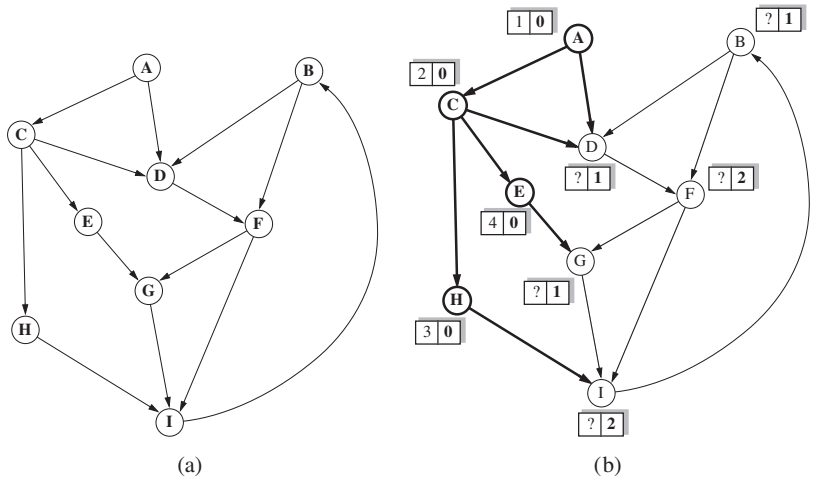


Figura 6.20 Detecção de um ciclo dirigido: (a) dígrafo de entrada; (b) depois que o algoritmo TopologicalSort (Algoritmo 6.19) termina, o subgrafo com os vértices sem número definido contém um ciclo dirigido.

Visualizamos o algoritmo de ordenação topológica na Figura 6.21.

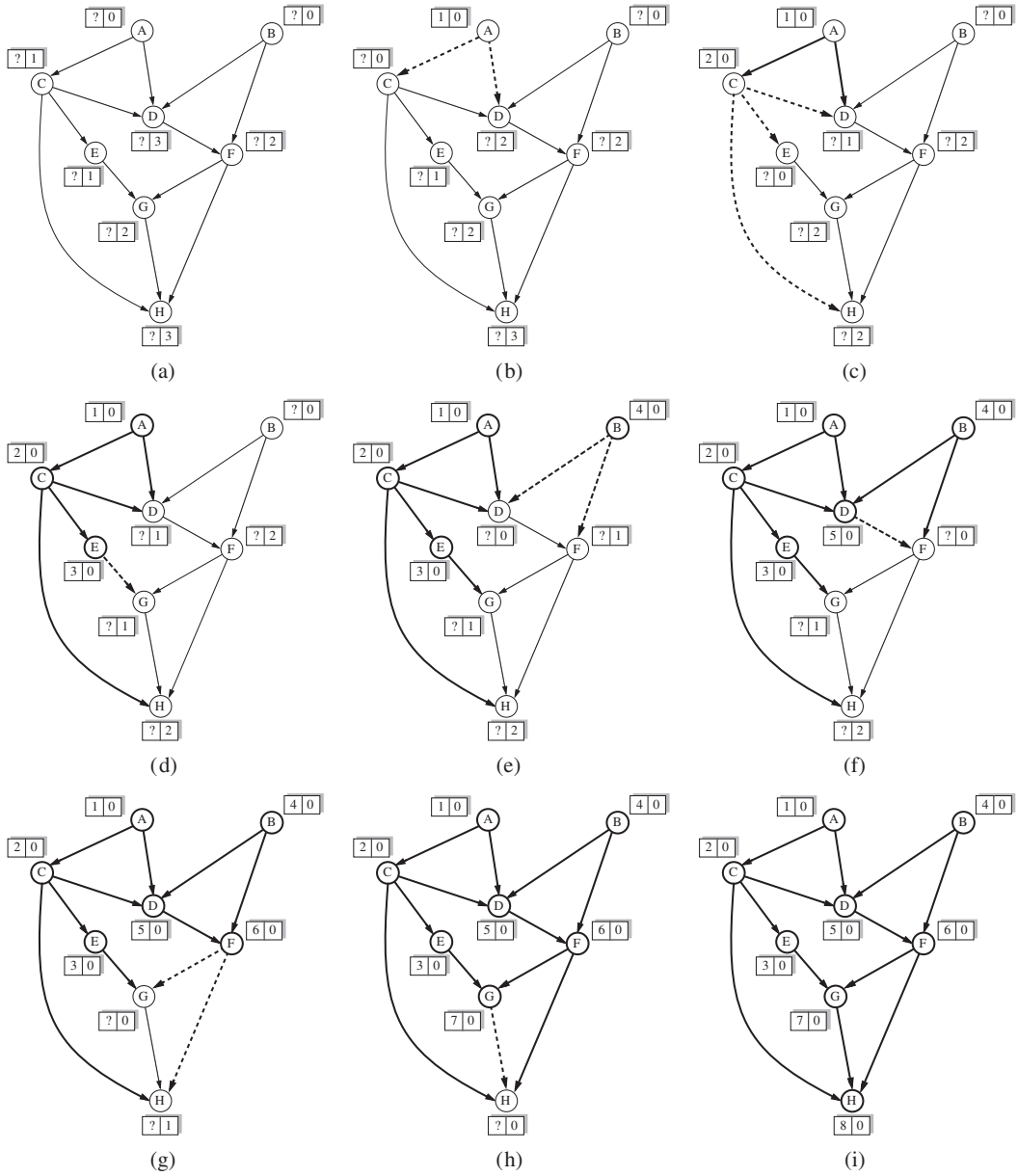


Figura 6.21 Exemplo de uma execução do algoritmo TopologicalSort (Algoritmo 6.19): (a) configuração inicial; (b – i) após cada iteração do laço **enquanto**. Os números nos vértices mostram o número do vértice e o valor corrente de *incounter* para o vértice. As arestas percorridas em iterações anteriores são mostradas com linhas espessas. Linhas pontilhadas espessas mostram as arestas examinadas na iteração corrente.

6.5 Exemplo em Java: caminhamento em profundidade

Nesta seção, mostramos um estudo de caso da implementação do caminhamento em profundidade em Java. Esta implementação não é uma simples instanciação do pseudocódigo em Java, no entanto, pois ela emprega dois interessantes padrões de projeto em engenharia de *software* – o padrão decorador e o método de *templates*.

6.5.1 O padrão decorador

Marcar os vértices explorados no caminhamento em profundidade é um exemplo do padrão ***decorador*** usado em engenharia de *software*. Esse padrão é usado para adicionar atributos ou "decorações" a objetos existentes. Cada atributo é identificado por um objeto específico a de decoração, que age como uma espécie de chave identificando o atributo. Intuitivamente, o atributo a é o "nome" da decoração, e permitimos que esse atributo assuma valores diferentes para os diferentes objetos que são decorados com atributos a. O uso de decorações é motivado pela necessidade de alguns algoritmos e estruturas de dados adicionarem variáveis extras ou dados temporários a objetos que normalmente não têm estas variáveis. Portanto, uma decoração é um par (atributo, valor) que pode ser associado dinamicamente a um objeto. Por exemplo, nodos em uma árvore AVL ou árvore vermelho-preta podem ser decorados com atributos de altura ou cor. Em nosso exemplo do caminhamento em profundidade, gostaríamos de ter vértices e arestas "decoráveis" com um atributo ***explored*** e um valor booleano.

Podemos realizar o padrão decorador para todos os contêineres posicionais adicionando os métodos seguintes para o TAD posição:

has(a): testa se a posição tem o atributo a.

get(a): retorna o valor do atributo a.

set(a,x): muda o valor do atributo a para x.

destroy(a): remove o atributo a e seu valor associado.

Implementação do padrão decorador

Os métodos acima podem ser implementados armazenando a cada posição um dicionário de atributos e seus valores para aquela posição, por exemplo, em uma tabela *hash*. Ou seja, os atributos, que podem ser objetos específicos ou cadeias de caracteres definindo "nomes" para atributos, definem as chaves usadas na tabela *hash*. Da mesma forma, os atributos são tipos básicos ou objetos que associam dados a atributos. Uma implementação assim de uma tabela *hash* fornece uma maneira rápida e eficiente de implementar os métodos do padrão decorador. De fato, embora não seja mostrado, usamos uma pequena tabela *hash* em nossa aplicação para implementar o padrão decorador.

6.5.2 Um mecanismo para caminhamento em profundidade

No Trecho de Código 6.22, mostramos o mecanismo Java principal para um caminhamento recursivo em profundidade encapsulado na classe DFS.

```
/** Caminhamento em profundidade genérico de um grafo usando
  * o padrão de template. Uma subclasse deve substituir vários métodos
  * para adicionar funcionalidade a este caminhamento. */
public abstract class DFS {
  /** O grafo sendo percorrido. */
  protected InspectableGraph G;
  /** O resultado do caminhamento. */
  protected Object visitResult;
  /** Realiza um caminhamento em profundidade.
    * @param g Grafo de entrada.
    * @param start Vértice inicial do caminhamento.
    * @param info Informação auxiliar (a ser usada pelas subclasses).
    */
  public Object execute(InspectableGraph g, Vertex start, Object info) {
    G = g;
    for (PositionIterator pos = G.positions(); pos.hasNext(); )
      unVisit(pos. next Position());
    return null;
  }
  /** Template recursivo para um caminhamento em profundidade genérico.
    * @param v Vértice inicial do caminhamento.
    */
  protected Object dfsTraversal (Vertex v) {
    initResult();
    startVisit(v);
    visit(v);
    for (EdgeIterator inEdges = G.incidentEdges(v); inEdges.hasNext();) {
      Edge nextEdge = in Edges.nextEdge();
      if (!isVisited (nextEdge)) { // achamos um vértice não explorado, e o exploramos
        visit(nextEdge);
        Vertex w = G.opposite(v, nextEdge);
        if (!isVisited(w)) { // w não foi explorado, esta é uma aresta de descoberta
          traverseDiscovery(nextEdge, v);
          if (!isDone())
            visitResult = dfsTraversal(w);
        }
        else // w foi explorado, esta é uma aresta de retorno }
          traverseBack(next Edge, v);
      }
    }
    finishVisit(v);
    return result();
  }
```

Trecho de Código 6.22 Variáveis principais e mecanismo (dfsTraversal) para a classe DFS.

6.5.3 O padrão de projeto com *templates*

A classe DFS é definida de forma que seu comportamento possa ser especializado para qualquer aplicação específica. Nos Trechos de Código 6.23 e 6.24, mostramos os métodos que realizam a especialização e decorações na classe DFS. A especialização é feita redefinindo-se o método execute, que ativa a computação, e os métodos a seguir, que são chamados em vários momentos pelo método recursivo baseado em templates dfsTraversal.

- initResult( ): chamado no início da execução de dfsTraversal.
- startVisit(Vertex *v*): também chamado no início da execução de dfsTraversal.
- traverseDiscovery(Edge *e*, Vertex *v*): chamado quando uma aresta de descoberta *e* saindo de *v* é percorrida.
- traverseBack(Edge *e*, Vertex *v*): chamado quando uma aresta de retorno *e* saindo de *v* é percorrida.
- isDone( ): chamado para determinar se devemos terminar o caminhamento antecipadamente.
- finishVisit(Vertex *v*): chamado quando todas as arestas incidentes a *v* foram percorridas.
- result( ): chamado para retornar o resultado de dfsTraversal.

```
/** Métodos auxiliares para especializar o DFS genérico */
protected void initResult( ) { } // Inicializa o resultado (chamado primeiro)
protected void startVisit(Vertex v) { } // Chamado quando visitamos v pela
  primeira vez
protected void finishVisit(Vertex v) { } // Chamado quando terminamos com v
protected void traverseDiscovery(Edge e, Vertex from) { } // Aresta de descoberta
protected void traverse Back(Edge e, Vertex from) { } // Aresta de retorno
protected boollean isDone( ) { return false; } // DFS já terminou?
protected Object result( ) { return new Object( ); } // Resultado do DFS
```

Trecho de Código 6.23 Métodos auxiliares para especializar o *template* dfsTraversal da classe DFS (Trecho de Código 6.22).

```
/** Atributo e seus dois valores para o status de visita às posições. */
protected static Object STATUS = new Object( ); // Atributo de status
protected static Object VISITED = new Object( ); // Valor visitado
protected static Object UNVISITED = new Object( ); // Valor não visitado
/** Marca posição visitada. */
protected void visit(Position p) { p.set(STATUS, VISITED); }
/** Marca posição não visitada. */
protected void unVisit(Position p) { p.set(STATUS, UNVISITED); }
/** Testa se uma posição foi visitada. */
protected boollean isVisited(Position p) { return (p.get(STATUS) == VISITED); }
```

Trecho de Código 6.24 Os métodos visit, unVisit e isVisited da classe DFS (Trecho de Código 6.22), implementados usando posições decoráveis.

Definição do padrão de *templates*

Nosso caminhamento em profundidade genérico é baseado no método de *templates*, que descreve um mecanismo de computação genérico que pode ser especializado redefinindo-se certos passos. Ou seja, definimos uma classe genérica, Algorithm, que realiza uma função útil e possivelmente complicada ao chamar uma coleção de procedimentos em determinados momentos. Inicialmente os procedimentos não fazem nada, mas ao estender a classe Algorithm e redefinir estes métodos para fazer coisas interessantes, podemos construir uma computação que não seja trivial. A classe Applet em Java é um exemplo do uso deste padrão de projeto. No caso de nossa aplicação para DFS, usamos os *templates* para caminhamento em profundidade assumindo que o grafo não é dirigido, mas ele pode ser modificado facilmente para operar sobre grafos dirigidos.

O mecanismo usado para identificar os vértices e arestas que já foram visitados (explorados) durante o caminhamento é encapsulado nas chamadas para os métodos isVisited, visit e unVisit. Nossa implementação (veja o Trecho de Código 6.24) assume que as posições dos vértices e arestas suportam o padrão decorador. (Alternativamente, podemos fazer um dicionário de posições e armazenar os vértices e arestas visitados nele.)

Usando o *template* para DFS

Para fazer algo interessante com dfsTraversal, devemos estender a classe DFS e redefinir alguns dos métodos auxiliares do Trecho de Código 6.23 para uma tarefa importante. Esta abordagem segue o padrão de métodos de *templates*, pois eles especializam o comportamento do método *template* dfsTraversal.

A classe ConnectivityTesterDFS (Trecho de Código 6.25), por exemplo, estende DFS para criar um programa que testa se um grafo é conexo. Ela conta o número de vértices atingíveis através de um caminhamento em profundidade iniciando em um vértice e compara esse número com o número total de vértices do grafo.

```
/** Esta classe especializa DFS para determinar se o grafo é conexo. */
public class ConnectivityTesterDFS extends DFS {
  protected int reached;
  public Object execute(InspectableGraph g, Vertex start, Object info) {
    super.execute(g, start, info);
    reached = 0;
    if (!G.isEmpty( )) {
      Vertex v = G.aVertex( );
      dfsTraversal(v);
    }
    return (new Boolean(reached == G.numVertices( )));
  }
  public void startVisit (Vertex v) { reached++; }
}
```

Trecho de Código 6.25 Especialização da classe DFS para testar se o grafo é conexo.

Estendendo DFS para determinar caminhos

Podemos obter algoritmos ainda mais interessantes estendendo DFS de maneiras mais engenhosas.

Por exemplo, a classe FindPathDFS dada no Trecho de Código 6.26 encontra um caminho entre um par de vértices dados e realiza um caminhamento em profundidade a partir do vértice inicial. Mantemos o caminho formado pelas arestas de descoberta a partir do vértice inicial até o vértice corrente. Quando encontramos um vértice inexplorado, ele é adicionado ao final do caminho e, quando terminamos de processar o vértice, o removemos do caminho. O caminhamento é terminado quando o vértice final é encontrado e o caminho é retornado como um iterador de vértices. Note que o caminho encontrado por esta classe se constitui de arestas de descoberta.

```
/** Esta classe especializa DFS para achar um caminho entre
  * um vértice de início e um de final. */
public class FindPathDFS extends DFS {
   protected Sequence path;
   protected boolean done;
   protected Vertex target;
   /** @param info vértice final do caminho
     * @return Iterador dos vértices em um caminho a partir do vértice
     * inicial ao vértice final ou um iterador vazio se não existir um
     * caminho*/
   public Object execute(InspectableGraph g, Vertex start, Object info) {
      super.execute(g, start, info);
      path = new NodeSequence( );
      done = false;
      target = (Vertex) info; // vértice final é armazenado no parâmetro info
      dfsTraversal(start);
      return new VertexIteratorAdapter(path. elements( ));
   }
   protected void startVisit(Vertex v) {
      path.insertLast(v);
      if (v == target)
         done = true;
   }
   protected void finishVisit(Vertex v) {
      if (!done)
         path. remove(path. last( ));
   }
   protected boolean isDone( ) {
      return done;
   }
}
```

Trecho de Código 6.26 Especialização da classe DFS para achar um caminho entre um vértice inicial e um vértice final. A classe auxiliar VertexIteratorAdapter fornece um adaptador a partir de uma seqüência para um iterador de vértices.

Estendendo DFS para determinação de ciclos

Podemos também estender DFS como na classe FindCycleDFS, dada no Trecho de Código 6.27, que encontra um ciclo no componente conexo de um dado vértice v. O algoritmo realiza um caminhamento em profundidade a partir de v que termina quando uma aresta de retorno é encontrada. Ela retorna um iterador (possivelmente vazio) do ciclo formado pela aresta de retorno encontrada.

```
/** Especializa DFS para achar um ciclo no componente conexo do vértice inicial. */
public class FindCycleDFS extends DFS {
  protected Sequence cycle; // seqüência de arestas do ciclo
  protected boolean done;
  protected Vertex cycleStart, target;
  /** @return iterador das arestas do ciclo no componente conexo do vértice inicial
    */
  public Object execute(InspectableGraph g, Vertex start, Object info) {
    super. execute(g, start, info);
    cycle = new NodeSequence( );
    done = false;
    dfsTraversal(start);
    if (!cycle.isEmpty( ) && start != cycleStart) {
      PositionIterator pos = cycle.positions( );
      while (pos.hasNext( )) { // remove as arestas do início até cycleStart
        Position p = pos.nextPosition( );
        Edge e = (Edge) p.element( );
        cycle. remove(p);
        if (g.areIncident(cycleStart, e)) break;
      }
    }
    return new EdgeIteratorAdapter(cycle.elements( ));
  }
  protected void finishVisit(Vertex v) {
    if ((!cycle. isEmpty( )) && (!done)) cycle.remove(cycle. last( ));
  }
  protected void traverseDiscovery(Edge e, Vertex from) {
    if (!done) cycle.insertLast(e);
  }
  protected void traverseBack(Edge e, Vertex from) {
    if (!done) {
      cycle.insertLast(e); // a aresta de retorno e cria um ciclo
      cycleStart = G.opposite (from, e);
      done = true;
    }
  }
  protected boolean isDone( ) { return done; }
}
```

Trecho de Código 6.27 Especialização da classe DFS para encontrar um ciclo no componente conexo do vértice inicial. A classe auxiliar EdgeIteratorAdapter provê um adaptador a partir de uma seqüência para um iterador de arestas.

6.6 Exercícios

Reforço

R-6.1 Desenhe um grafo simples não-dirigido G com 12 vértices, 18 arestas e três componentes conexos. Por que seria impossível desenhar G com três componentes conexos se G tivesse 66 arestas?

R-6.2 Seja G um grafo simples conexo com n vértices e m arestas. Explique por que $O(\log(m))$ é $O(\log(n))$.

R-6.3 Desenhe um grafo simples, conexo e dirigido G com 8 vértices e 16 arestas de forma que o grau de entrada e de saída de cada vértice seja 2. Mostre que existe um único ciclo (não-simples) que inclui todas as arestas do grafo, ou seja, que você pode desenhar todas as arestas em suas direções respectivas sem levantar o lápis do papel (este tipo de ciclo é chamado de ***ciclo de Euler***).

R-6.4 Bob adora línguas estrangeiras e deseja planejar seus horários para os próximos anos. Ele está interessado nos seguintes nove cursos de idiomas: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141 e LA169. Os pré-requisitos dos cursos são:

- LA15: nenhum
- LA16: LA15
- LA22: nenhum
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32.

Ache a seqüência de cursos que permite que Bob satisfaça todos os pré-requisitos.

R-6.5 Suponha que representamos um grafo G com n vértices e m arestas com uma lista de arestas. Por que, neste caso, o método insertVertex é executado em tempo $O(1)$ enquanto o método removeVertex leva tempo $O(m)$?

R-6.6 Seja G um grafo cujos vértices são os inteiros de 1 a 8 e sejam os vértices adjacentes a cada vértice dados pela tabela abaixo:

vértice	vértice adjacente
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Suponha que em um caminhamento de G os vértices adjacentes a um vértice dado são retornados na mesma ordem em que são listados na tabela acima.

a. Desenhe G.
b. Descreva a seqüência de vértices de G visitada usando um caminhamento em profundidade, iniciando no vértice 1.
c. Descreva a seqüência de vértices de G visitada usando um caminhamento em largura, iniciando no vértice 1.

R-6.7 Você usaria uma lista de adjacência ou uma matriz de adjacência nos casos abaixo? Justifique sua escolha.

a. O grafo tem 10.000 vértices e 20.000 arestas e é importante usar o menor espaço possível.
b. O grafo tem 10.000 vértices e 20.000.000 arestas e é importante usar o menor espaço possível.
c. Você deve responder a uma consulta areAdjacent tão rapidamente quanto possível, sem se importar com quanto espaço você usa.

R-6.8 Explique por que o caminhamento em profundidade tem tempo $\Theta(n^2)$ em um grafo simples de n vértices representado com uma lista de adjacência.

R-6.9 Desenhe o fechamento transitivo do grafo dirigido da Figura 6.2.

R-6.10 Determine uma ordenação topológica para o grafo dirigido desenhado com arestas sólidas na Figura 6.14d.

R-6.11 Podemos usar uma fila em vez de uma pilha como estrutura de dados auxiliar no algoritmo de ordenação topológica mostrado no Fragmento de Código 6.19?

R-6.12 Descreva a ordem em que as arestas são visitadas pelo caminhamento em profundidade mostrado na Figura 6.6.

R-6.13 Repita o Exercício R-6.12 para a Figura 6.13 ilustrando um caminhamento em largura.

R-6.14 Repita o Exercício R-6.12 para a Figura 6.15 ilustrando um caminhamento em profundidade dirigido.

Criatividade

C-6.1 Justifique o Teorema 6.11.

C-6.2 Descreva os detalhes de um algoritmo de tempo $O(n + m)$ para determinar ***todos*** os componentes conexos de um grafo não-dirigido com n vértices e m arestas.

C-6.3 Seja T a árvore de cobertura com raiz no vértice inicial produzido pelo caminhamento em profundidade em um grafo G conexo e não-dirigido. Justifique por que toda aresta de G fora de T vai de um vértice em T a um de seus predecessores, ou seja, é uma ***aresta de retorno***.

Dica: suponha que esta aresta é uma aresta de cruzamento e argumente baseado na ordem de visitação do caminhamento em profundidade para os vértices finais desta aresta.

C-6.4 Suponha que desejamos representar um grafo de n vértices usando uma lista de arestas, e identificamos os vértices com os inteiros do conjunto $\{0, 1, ..., n-1\}$. Descreva como implementar o contêiner E para suportar desempenho $O(\log(n))$ para o método areAdjacent. Como você vai implementar o método neste caso?

C-6.5 Forneça uma prova do Lema 6.14.

C-6.6 Demonstre que, se um grafo G tem pelo menos três vértices, então ele tem uma aresta de separação somente se ele tiver um vértice de separação.

C-6.7 Forneça uma prova do Lema 6.16.

C-6.8 Forneça os detalhes da prova de correção do algoritmo LinkComponents (Algoritmo 6.10).

C-6.9 A Universidade de Tamarindo e várias outras escolas ao redor do mundo estão envolvidas em um projeto multimídia. Uma rede de computadores é montada para conectar essas escolas usando linhas de comunicação que formam uma árvore livre. Decide-se instalar um servidor de arquivos em uma das escolas para compartilhar dados entre todas elas. Como o tempo de transmissão em uma conexão é controlado por sua inicialização e sincronização, o custo da transferência de dados é proporcional ao número de conexões usadas. Assim, é desejável escolher uma escola "central" para o servidor. Dada uma árvore livre T e um nodo v de T, a *excentricidade* de v é o comprimento do maior caminho de v a qualquer outro nodo de T. Um nodo de T com a menor excentricidade é chamado de *centro* de T.

a. Projete um algoritmo eficiente que, dada uma árvore livre T com n nodos, determine seu centro.
b. O centro é único? Se não for, quantos centros diferentes uma árvore livre pode ter?

C-6.10 Demonstre como realizar o caminhamento em largura usando como estrutura de dados auxiliar uma única fila em vez dos contêineres $L_0, L_1, ...$

C-6.11 Demonstre que, se T é uma árvore produzida por caminhamento em largura para um grafo conexo G, então, para cada vértice v no nível i, o caminho de T entre s e v tem i arestas e qualquer outro caminho de G entre s e v tem pelo menos i arestas.

C-6.12 O atraso em uma chamada de longa distância pode ser determinado multiplicando-se uma pequena constante pelo número de conexões telefônicas entre os pontos sendo ligados. Suponha que a rede telefônica da companhia RT&T é uma árvore livre. Os engenheiros da RT&T desejam avaliar o maior atraso possível em uma chamada de longa distância. Dada uma árvore livre T, o *diâmetro* de T é o comprimento do caminho mais longo entre dois nodos de T. Forneça um algoritmo eficiente para determinar o diâmetro de T.

C-6.13 Uma companhia chamada RT&T tem uma rede de n estações telefônicas conectadas por m linhas de alta velocidade. O telefone de cada cliente é conectado diretamente a uma estação em sua área. Os engenheiros da RT&T desenvolveram um protótipo de videofone que permite que dois clientes vejam um ao outro durante uma chamada. Para ter uma imagem de qualidade aceitável, no entanto, o número de conexões usadas para transmitir os sinais de vídeo entre as partes não pode exceder a 4. Suponha que a rede da RT&T é representada por um grafo. Planeje um algoritmo eficiente que determina, para cada estação, o conjunto de estações que podem ser alcançadas com 4 conexões ou menos.

C-6.14 Explique por que não existem arestas de descoberta fora da árvore produzida por um caminhamento em largura construído para um grafo dirigido.

C-6.15 Forneça uma descrição detalhada em pseudocódigo do algoritmo de caminhamento em profundidade dirigido. Como determinamos se uma aresta fora da árvore é uma aresta de retorno, aresta de descoberta ou aresta de cruzamento?

C-6.16 Explique por que o algoritmo de teste de conectividade forte dado na Seção 6.4.1 está correto.

C-6.17 Seja G um grafo não-dirigido com n vértices e m arestas. Descreva um algoritmo de tempo de execução $O(n + m)$ que percorre cada aresta de G exatamente uma vez em cada direção.

C-6.18 Um conjunto independente de um grafo não-dirigido $G = (V,E)$ é um subconjunto I de V tal que dois vértices em I não são adjacentes. Ou seja, se $u,v \in I$, então $(u,v) \notin E$. Um ***conjunto máximo independente*** M é um conjunto independente que, se incluíssemos qualquer vértice adicional a M, então ele não seria mais um conjunto independente. Todo grafo tem um conjunto máximo independente. (Você percebe isso? Essa questão não faz parte do exercício, mas vale a pena pensar sobre ela.) Forneça um algoritmo eficiente que determina um conjunto máximo independente para um grafo G. Qual o tempo de execução do método?

C-6.19 Um ***ciclo de Euler*** de um grafo dirigido $\vec{G}$ com n vértices e m arestas é um ciclo que passa por cada aresta de $\vec{G}$ exatamente um vez de acordo com sua direção. Um ciclo desse tipo sempre existe se $\vec{G}$ for conexo e o grau de entrada for igual ao grau de saída para cada vértice em $\vec{G}$. Descreva um algoritmo de tempo $O(n + m)$ para achar um ciclo de Euler em um dígrafo $\vec{G}$.

C-6.20 Justifique o Teorema 6.18.

Projetos

P-6.1 Escreva uma classe implementando um TAD simplificado para grafos que tenha os métodos relevantes para grafos não-dirigidos e não inclui métodos de atualização usando uma matriz de adjacência.

P-6.2 Implemente o TAD simplificado descrito no Projeto P-6.1 usando uma lista de adjacência.

P-6.3 Implemente o caminhamento em profundidade genérico usando o padrão de *templates*.

P-6.4 Implemente o algoritmo de ordenação topológica.

P-6.5 Implemente o algoritmo de Floyd-Warshall para o fechamento transitivo.

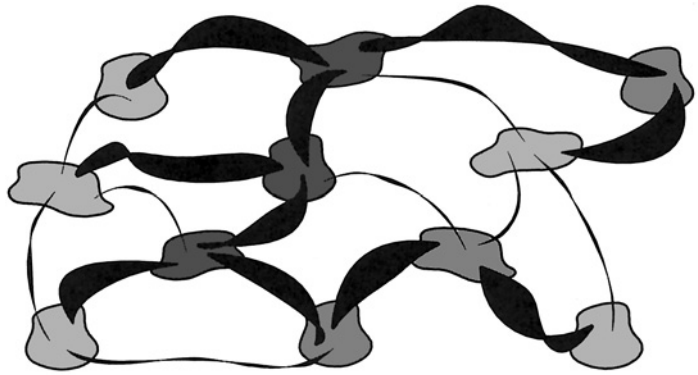
Notas

O método de caminhamento em profundidade é parte do folclore da Computação, mas Hopcroft e Tarjan [98,198] foram os que mostraram o quanto este algoritmo é útil para resolver vários problemas diferentes de grafos. Knuth [118] discute o problema da ordenação topológica. O algoritmo de tempo linear simples que descrevemos na Seção 6.4.1 para determinar se um grafo dirigido é fortemente conexo é de Kosaraju. O algoritmo de Floyd-Warshall é descrito em um artigo de Floyd [69] e é baseado em um teorema de Warshall [209]. O método de coleta de lixo por marcação e varredura é um dos muitos algoritmos para coleta de lixo. Encorajamos o leitor interessado no estudo da coleta de lixo a consultar o livro de Jones [110]. Para aprender sobre diferentes métodos para desenho de grafos, veja o capítulo de Tamassia [194], a bibliografia comentada de Di Battista *et al*. [58] ou o livro de Di Battista *et al*. [59].

O leitor interessado em um estudo mais profundo sobre grafos pode consultar os livros de Ahuja, Magnanti e Orlin [9], Cormen, Leiserson e Rivest [55], Even [68], Gibbons [77], Mehlhorn [149] e Tarjan [200] e o capítulo de van Leeuwen [205].

Capítulo

7 Grafos ponderados



7.1 Caminhos mínimos a partir de um vértice 344

7.1.1 O algoritmo de Dijkstra 345

7.1.2 O algoritmo de Bellman-Ford para caminhos mínimos 351

7.1.3 Caminhos mínimos em grafos acíclicos dirigidos 354

7.2 Caminhos mínimos entre todos os vértices 356

7.2.1 Um algoritmo de caminho mínimo com programação dinâmica 356

7.2.2 Determinação dos caminhos mínimos através da multiplicação de matrizes 357

7.3 Árvores de cobertura mínima 361

7.3.1 O algoritmo de Kruskal 363

7.3.2 O algoritmo de Prim-Jarník. 367

7.3.3 O algoritmo de Barůvka 369

7.3.4 Comparação entre os algoritmos 372

7.4 Exemplo em Java: o algoritmo de Dijkstra 372

7.5 Exercícios 376

Como vimos no capítulo anterior, a estratégia de caminhamento em largura pode ser usada para encontrarmos um caminho mínimo de um vértice inicial a cada um dos outros vértices em um grafo conexo. Essa abordagem faz sentido em casos em que cada aresta é tão boa quanto qualquer outra, mas existem muitas situações onde ela não é apropriada.

Por exemplo, podemos estar usando um grafo para representar uma rede de computadores (como a Internet) e estar interessados em encontrar o caminho mais rápido para enviar um pacote de dados entre dois computadores. Neste caso, provavelmente não é correto considerar todas as arestas equivalentes, pois algumas conexões na rede são tipicamente muito mais rápidas do que outras (por exemplo, algumas arestas podem representar conexões lentas por linha telefônica enquanto outras representam ligações de alta velocidade em fibra ótica). Da mesma forma, se desejarmos usar um grafo para representar as estradas entre cidades, podemos estar interessados em achar as distâncias mais curtas entre elas. Neste caso, provavelmente também não é correto considerar todas as arestas equivalentes, pois algumas distâncias serão muito maiores do que outras. Assim, é natural considerar grafos cujas arestas não têm pesos equivalentes.

Neste capítulo, estudaremos grafos ponderados. Um ***grafo ponderado*** é um grafo que tem um valor numérico $w(e)$ associado a cada aresta e e chamado de ***peso*** de e. Os pesos podem ser inteiros, números racionais ou números reais, que representam conceitos como distâncias, custos de conexão ou afinidade. Mostramos um exemplo de um grafo ponderado na Figura 7.1.

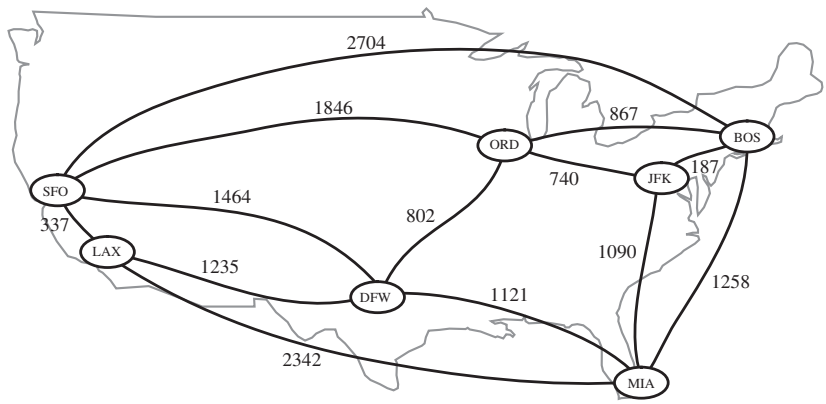


Figura 7.1 Um grafo ponderado cujos vértices representam aeroportos norte-americanos e cujos pesos das arestas representam distâncias (em milhas). Este grafo tem um caminho de JFK para LAX com peso total 2.777 (passando por ORD e DFW). Este é o caminho com peso mínimo no grafo entre JFK e LAX.

7.1 Caminhos mínimos a partir de um vértice

Seja G um grafo ponderado. O ***comprimento*** (ou ***peso***) de um caminho P é a soma dos pesos das arestas de P. Ou seja, se P consiste nas arestas $e_0, e_1, ..., e_{k-1}$, então o comprimento de P, denotado por $w(P)$, é definido como

$$w(P) = \sum_{i=0}^{k-1} w(e_i).$$

A ***distância*** de um vértice v a um vértice u em G, denotada por $d(v,u)$, é o comprimento de um caminho de comprimento mínimo (também chamado ***caminho mínimo***) de v para u, se um tal caminho existir.

Por vezes usa-se a convenção $d(v,u) = +\infty$ se não houver caminho de v para u em G. Mas mesmo se houver um caminho de v para u em G, a distância de v para u pode não estar definida se existir um ciclo em G que tenha comprimento negativo. Por exemplo, suponha que os vértices em G representam cidades, e os pesos das arestas em G representam quanto dinheiro custa viajar de uma cidade a outra. Se desejassem nos pagar para viajar de JFK para ORD, então o custo da aresta (JFK, ORD) seria negativo. Se estivéssemos sendo pagos para viajar de ORD para JFK, teríamos um ciclo negativo em G, e as distâncias deixariam de estar definidas. Ou seja, qualquer um pode construir um caminho (com ciclos) em G de qualquer cidade A a outra cidade B passando por JFK e fazendo ciclos entre JFK e ORD tanto quanto se queira até seguir para B. A existência desses caminhos permite-nos construir caminhos com custo negativo arbitrariamente baixo (e ficar ricos em nosso exemplo). Mas distâncias não podem ser números negativos arbitrários. Assim, sempre que usamos pesos de arestas para representar distâncias, devemos ter cuidado para não introduzir ciclos de comprimento negativo.

Suponha que temos um grafo ponderado G e somos solicitados a encontrar um caminho mínimo de um vértice v a cada outro vértice em G, considerando o pesos das arestas como distâncias. Nesta seção, exploramos formas eficientes de encontrar todos os caminhos mínimos, se existirem.

O primeiro algoritmo que discutiremos é para o caso simples e comum em que todos os pesos das arestas em g são não-negativos (ou seja, $w(e) \geq 0$ para toda aresta e de G); portanto, sabemos de antemão que não pode haver ciclos negativos em G. Lembre que o caso especial de determinar um caminho mínimo quando todos os pesos forem 1 foi resolvido com o algoritmo de caminhamento em largura apresentado na Seção 6.3.3.

Existe uma abordagem interessante para resolver este problema de ***origem única*** baseado no ***método guloso*** (Seção 5.1). Lembre que com este método resolvemos o problema fazendo repetidamente a melhor escolha entre as disponíveis em cada iteração. Este paradigma pode ser freqüentemente usado em situações em que tentamos otimizar alguma função de custo em uma coleção de objetos. Podemos adicionar um objeto de cada vez à nossa coleção, sempre escolhendo o próximo que otimiza a função entre aqueles objetos ainda a serem escolhidos.

7.1.1 O algoritmo de Dijkstra

A idéia principal na aplicação do método guloso para o problema do caminho mínimo com origem única é realizar um caminhamento em largura "ponderado", iniciando-se em v. Especificamente, podemos usar o método guloso para desenvolver um algoritmo que iterativamente aumente uma "nuvem" de vértices em torno de v, com os vértices entrando na nuvem na seqüência de suas distâncias de v. Assim, a cada iteração, o próximo vértice escolhido é o vértice fora da nuvem e que está mais próximo de v. O algoritmo termina quando não há mais vértices fora da nuvem e, neste momento, temos o caminho mais curto de v para qualquer outro vértice de G. Esta abordagem é um exemplo simples e poderoso do padrão de projeto baseado no método guloso.

Um método guloso para determinar caminhos mínimos

Aplicar o método guloso para o problema do caminho mínimo com origem única resulta em um algoritmo conhecido como ***algoritmo de Dijkstra***. Quando aplicado a outros problemas de grafos, no entanto, o método guloso pode não achar necessariamente a melhor solução (como no caso do ***problema do caixeiro viajante***, no qual desejamos encontrar o menor caminho que visite todos os vértices do grafo exatamente uma vez). Mesmo assim, existem várias situações nas quais o método guloso permite-nos determinar a melhor solução. Neste capítulo, discutiremos duas destas situações: determinar caminhos mínimos e construir uma árvore de cobertura mínima.

Para simplificar a descrição do algoritmo de Dijkstra, assumiremos que o grafo de entrada G seja não-dirigido (ou seja, todas as suas arestas são não-dirigidas) e simples (ou seja, ele não tem arestas paralelas nem vértices com arestas para si mesmos). Assim, denotamos as arestas de G como pares de vértices não-ordenados (u, z). Deixamos a descrição do algoritmo de Dijkstra adaptado para um grafo ponderado e dirigido como exercício (R-7.2).

No algoritmo de Dijkstra, a função de custo que desejamos otimizar em nossa aplicação do método guloso também é a função que desejamos avaliar – a distância do caminho mínimo. Isso pode parecer um raciocínio circular até notarmos que podemos implementar esta abordagem usando um truque para inicializá-la que consiste em usar uma aproximação para a função de distância que desejamos calcular e que, ao final do processo, será exatamente igual à distância real.

Relaxamento de arestas

Vamos definir um rótulo $D[u]$ para cada vértice u em V, o qual usaremos para aproximar a distância em G de v até u. O significado destes rótulos é que $D[u]$ irá sempre armazenar o comprimento do melhor caminho que encontramos ***até o momento*** de v até u. Inicialmente, $D[v] = 0$ e $D[u] = +\infty$ para cada $u \neq v$, e definimos o conjunto C que servirá como nossa *"**nuvem**"* de vértices, inicializado como vazio. A cada iteração do algoritmo, selecionamos um vértice u que não está em C com o menor rótulo $D[u]$ e colocamos u em C. Na primeira iteração, iremos colocar v em C. Quando um novo vértice u é colocado em C, atualizamos o rótulo $D[z]$ de cada vértice z que é adjacente a u e não está em C, refletindo o fato de que pode haver uma nova e melhor maneira de chegar a z via u. Essa operação de atualização é conhecida como ***relaxamento***, pois recebe a antiga estimativa e verifica se ela pode ser melhorada para se aproximar de seu

valor verdadeiro. (Uma metáfora para o relaxamento é uma mola, que ao ser esticada e solta "relaxa" para sua posição natural de repouso). No caso do algoritmo de Dijkstra, o relaxamento é realizado para uma aresta (u,z) de forma que calculamos um novo valor para $D[u]$ e desejamos verificar se há um valor melhor para $D[z]$ usando a aresta (u,z). A operação específica de relaxamento de uma aresta é a que segue:

Relaxamento de uma aresta:

se $D[u] + w((u, z)) < D[z]$ **então**
$D[z] \leftarrow D[u] + w((u, z))$

Note que, se o novo caminho encontrado para z não for melhor do que o caminho antigo, então não mudamos o valor de $D[z]$.

Detalhes do algoritmo de Dijkstra

Fornecemos o pseudocódigo para o algoritmo de Dijkstra no Algoritmo 7.2. Note que usamos uma fila de prioridade Q para armazenar os vértices fora da nuvem C.

Algoritmo DijkstraShortestPath(G,v):
Entrada: um grafo simples, ponderado e não-dirigido G e um vértice v de G.
Saída: um rótulo $D[u]$ para cada vértice u de G tal que $D[u]$ é o comprimento do caminho mínimo de v para u em G.
$D[v] \leftarrow 0$
para $u \neq v$ em $\vec{G}$ **faça**
$D[u] \leftarrow +\infty$.
Crie uma fila de prioridade Q contendo todos os vértices de G usando D rótulos como chave.
enquanto Q não estiver vazia **faça**
{coloque um novo vértice u na nuvem}
$u \leftarrow Q$.removeMin()
para cada vértice z adjacente a u tal que z está em Q **faça**
{realizar o *relaxamento* na aresta (u,z)}
se $D[u] + w((u, z)) < D[z]$ **então**
$D[z] \leftarrow D[u] + w((u, z))$
Altere para $D[z]$ a chave do vértice z em Q
retorne o rótulo $D[u]$ de cada vértice u em G

Algoritmo 7.2 Algoritmo de Dijkstra para o problema do caminho mínimo com origem única para um grafo G iniciando de um vértice v.

Ilustramos várias iterações do algoritmo de Dijkstra nas Figuras 7.3 e 7.4.

Por que ele funciona

O aspecto interessante e um tanto surpreendente do algoritmo de Dijkstra é que, no momento em que um vértice u é colocado em C, seu rótulo $D[u]$ armazena o comprimento correto do caminho mínimo de v para u. Assim, quando o algoritmo termina, ele terá avaliado o comprimento do caminho mínimo de v para todos os vértices de G. Ou seja, ele terá resolvido o problema do caminho mínimo com origem única.

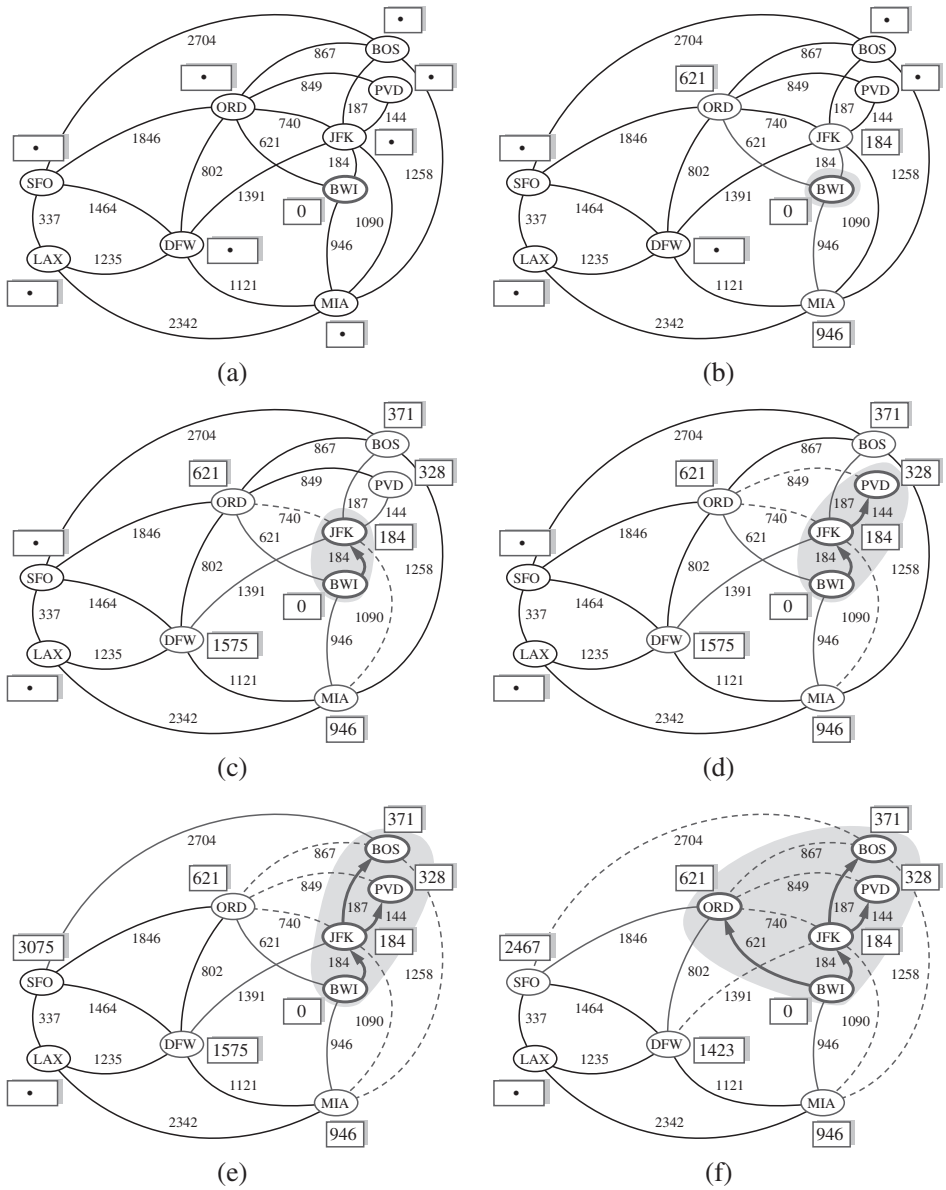


Figura 7.3 Execução do algoritmo de Dijkstra em um grafo ponderado. O vértice inicial é BWI. Uma caixa próxima a cada vértice v armazena o rótulo $D[v]$. O símbolo • é usado no lugar de $+\infty$. As arestas da árvore do caminho mínimo são desenhadas como setas espessas e, para cada vértice u fora da "nuvem", mostramos a melhor aresta atual para u com uma linha sólida. (Continua na Figura 7.4.)

É provável que não seja imediatamente claro o motivo pelo qual o algoritmo de Dijkstra encontra corretamente o caminho mínimo do vértice inicial v a um vértice u do grafo. Por que a distância de v a u é igual ao valor do rótulo $D[u]$ quando o vértice u é colocado em C (ou seja, quando u é removido da fila de prioridade Q)? A resposta desta questão depende da inexistência de arestas com pesos negativos no grafo, pois isto permite que o método guloso funcione corretamente, como mostraremos no lema que segue.

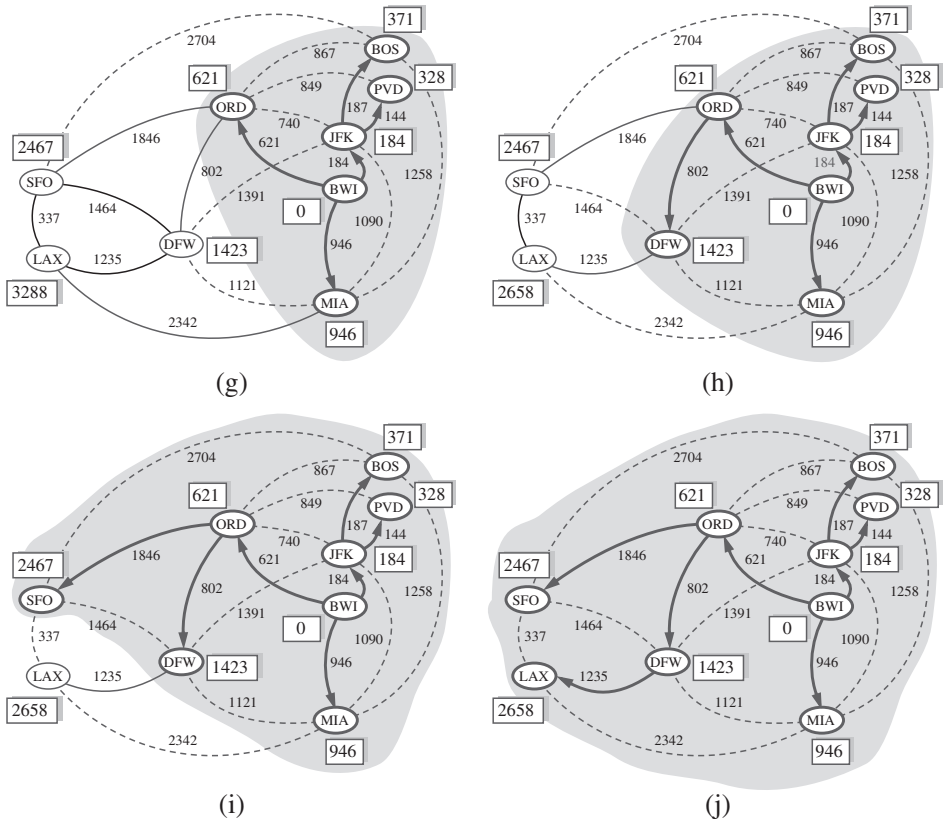


Figura 7.4 Execução do algoritmo de Dijkstra (continuação da Figura 7.3).

Lema 7.1: *no algoritmo de Dijkstra, sempre que um vértice u é colocado na nuvem, o rótulo* $D[u]$ *é igual a* $d(v, u)$ *– o comprimento do caminho mínimo de v para u.*

Prova: suponha que $D[t] > d(v, t)$ para algum vértice t em V, e seja u o ***primeiro*** vértice que o algoritmo colocou na nuvem C (ou seja, removeu de Q), tal que $D[u] > d(v, u)$. Existe um caminho mínimo P de v para u (pois, de outra forma, $d(v, u) = +\infty = D[u]$). Vamos considerar o momento em que u é colocado em C, e seja z o primeiro vértice de P (indo de v para u) que não está em C neste momento. Seja y o predecessor de z no caminho P (note que podemos ter $y = v$). (Veja a Figura 7.17.) Sabemos, por nossa escolha de z, que y já está em C neste momento. Além disso, $D[y] = d(v, y)$ pois u é o ***primeiro*** vértice incorreto. Quando y foi colocado em C, testamos (e possivelmente atualizamos) $D[z]$ de modo que tivemos naquele momento

$$D[z] \leq D[y] + w((y, z)) = d(v, y) + w((y, z)).$$

Mas já que z é o próximo vértice no caminho mínimo de v para u, isso implica que

$$D[z] = d(v, z).$$

Mas estamos agora escolhendo u e não z para ser colocado em C; por isso,

$$D[u] \leq D[z].$$

Deveria estar claro que um subcaminho de um caminho mínimo é também um caminho mínimo. Portanto, já que z está no caminho mínimo de v até u,

$$d(v, z) + d(z, u) = d(v, u).$$

Além disso, $d(z, u) \geq 0$, porque não há arestas com pesos negativos. Assim,

$$D[u] \leq D[z] = d(v, z) \leq d(v, z) + d(z, u) = d(v, u).$$

Mas isso contradiz a definição de u; portanto, não pode existir tal vértice u. ■

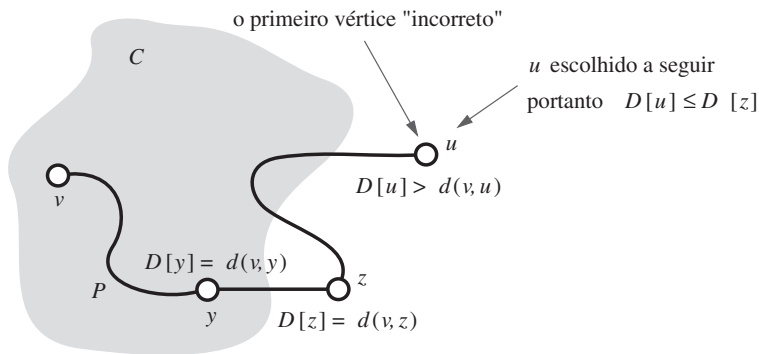


Figura 7.5 Esquema para a prova do Lema 7.1.

O tempo de execução do algoritmo de Dijkstra

Nesta seção, analisamos a complexidade do algoritmo de Dijkstra. Indicamos com n e m o número de vértices e arestas do grafo G, respectivamente. Assumimos que os pesos das arestas podem ser somados e comparados em tempo constante. Por causa do alto nível da descrição que fornecemos para o algoritmo de Dijkstra no Algoritmo 7.2, analisar seu tempo de execução requer que tenhamos mais detalhes de sua implementação. Especificamente, devemos indicar as estruturas de dados usadas e como elas são implementadas.

Vamos assumir que estamos representando o grafo G com uma lista de adjacência. Esta estrutura de dados permite-nos percorrer os vértices adjacentes a u durante a etapa de relaxamento em tempo proporcional a seu número. Isso ainda não decide todos os detalhes do algoritmo, pois devemos saber mais sobre como implementamos a outra estrutura de dados principal do algoritmo – a fila de prioridade Q.

Uma implementação eficiente da fila de prioridade Q usa um *heap* (veja a Seção 2.4.3), o que nos permite extrair o vértice u com o menor rótulo D chamando o método removeMin em tempo $O(\log(n))$. Como visto no pseudocódigo, cada vez que atualizamos o rótulo $D[z]$, temos de atualizar a chave de z na fila de prioridade. Se Q é implementada como um *heap*, então esta atualização da chave pode, por exemplo, ser feita retirando e em seguida reinserindo z com a nova prioridade. Se a fila de prioridade Q suporta o padrão de localizadores (veja a Seção 2.4.4), então podemos implementar facilmente estas atualizações de chave em tempo $O(\log(n))$, pois um localizador para

o vértice z permitiria que Q tivesse acesso direto ao item armazenando z no *heap* (veja a Seção 2.4.4). Assumindo esta implementação de Q, o algoritmo de Dijkstra é executado em tempo $O((n + m)\log(n))$.

Voltando ao Algoritmo 7.2, os detalhes da análise do tempo de execução são os seguintes:

- inserir todos os vértices em Q com suas chaves iniciais pode ser feito em tempo $O(n \log(n))$ através de inserções repetidas, ou tempo $O(n)$ usando a construção *bottom-up* (veja a Seção 7.3.5);
- a cada iteração do laço **enquanto**, gastamos tempo $O(\log(n))$ para remover o vértice u de Q e tempo $O(deg(v)\log(n))$ para realizar o relaxamento nas arestas incidentes a u;
- o tempo total de execução do laço **enquanto** é

$$\sum_{v \in G} (1 + deg(v)) \log n,$$

que é $O((n + m)\log(n))$ pelo Teorema 6.6.

Assim, temos o que segue:

Teorema 7.2: dado um grafo G ponderado com n vértices e m arestas, cada uma com um peso não-negativo, o algoritmo de Dijkstra pode ser implementado para encontrar todos os caminhos mínimos a partir de um vértice v em G em tempo $O(m \log(n))$.

Note que, se desejamos expressar o tempo de execução em função de n apenas, então ele é $O(n^2\log(n))$ no pior caso, já que assumimos que G é simples.

Uma implementação alternativa do algoritmo de Dijkstra

Vamos considerar uma implementação alternativa para a fila de prioridade Q usando uma seqüência não-ordenada. Isto, é claro, requer que gastemos tempo $\Omega(n)$ para retirar o menor elemento, mas permite atualizações de chave muito rápidas desde que Q suporte o padrão de localizadores (Seção 2.4.4). Especificamente, podemos implementar cada atualização de chave feita em uma etapa de relaxamento em tempo $O(1)$ – simplesmente alteramos o valor da chave depois de localizar o item em Q. Portanto, esta implementação resulta em um tempo de execução que é $O(n^2 + m)$, que pode ser simplificado para $O(n^2)$, já que G é simples.

Comparação das duas implementações

Temos duas escolhas para implementar a fila de prioridade no algoritmo de Dijkstra: uma implementação de *heap* baseada em localizadores, que tem tempo de execução $O(m \log(n))$, e uma implementação de seqüência não-ordenada baseada em localizadores, que tem tempo de execução $O(n^2)$. Já que ambas as implementações seriam relativamente simples de codificar, elas são aproximadamente iguais em termos de sofisticação da programação requerida. Estas duas implementações também são aproximadamente equivalentes em função dos fatores constantes em seus tempos de execução de pior caso. Olhando somente para esses tempos de execução de pior caso, preferimos a implementação baseada em *heap* quando o número de arestas em um grafo é pequeno (ou seja, quando $m < n^2/\log(n)$) e a implementação com seqüência quando o número de arestas é grande (ou seja, quando $m > n^2/\log(n)$).

Teorema 7.3: dado um grafo simples não-dirigido ponderado G com n vértices e m arestas tal que o peso de cada aresta é não-negativo e um vértice v de G, o algoritmo de Dijkstra determina a distância de v para todos os outros vértices de G em tempo $O(m \log(n))$, ou alternativamente em tempo $O(n^2)$.

No Exercício R-7.3, exploraremos como modificar o algoritmo de Dijkstra para produzir uma árvore T com raiz em v tal que o caminho em T do vértice v a um vértice u é o caminho mínimo em G de v para u. Além disso, adaptar o algoritmo de Dijkstra para grafos dirigidos é razoavelmente simples. No entanto, não podemos adaptar o algoritmo de Dijkstra para operar sobre grafos com ciclos negativos, como mostra a Figura 7.6.

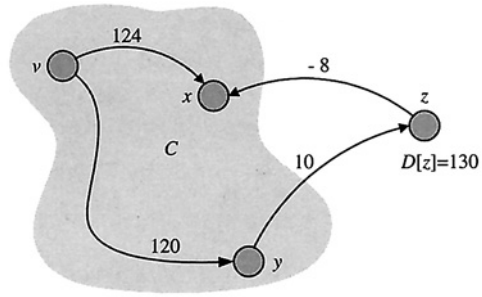


Figura 7.6 Uma ilustração de como o algoritmo de Dijkstra falha para grafos com pesos negativos. Colocar z em C e fazer o relaxamento irá invalidar o menor caminho calculado anteriormente (124) até x.

7.1.2 O algoritmo de Bellman-Ford para caminhos mínimos

Existe outro algoritmo, desenvolvido por Bellman e Ford, que encontra caminhos mínimos em grafos que contêm pesos negativos em suas arestas. Neste caso, devemos insistir para que o grafo seja dirigido, pois, se ele não for, qualquer aresta com peso negativo originará um ciclo negativo se for percorrida repetidas vezes nas duas direções. Não podemos permitir essas arestas, pois um ciclo negativo invalida a noção de distância baseada nos pesos das arestas.

Seja $\vec{G}$ um grafo ponderado e dirigido, possivelmente com arestas de peso negativo. O algoritmo de Bellman-Ford para determinação de caminhos mínimos a partir de um vértice v em $\vec{G}$ para todos os outros vértices em $\vec{G}$ é bastante simples. Ele usa a noção de relaxamento das arestas, como o método de Dijkstra, mas não usa o método guloso (que não funciona neste contexto; veja o exercício C-7.2). Ou seja, como no algoritmo de Dijkstra, o algoritmo de Bellman-Ford usa um rótulo $D[u]$ que é sempre um limite superior para a distância $d(v, u)$ de v até u, que é iterativamente "relaxada" até igualar esta distância.

Os detalhes do algoritmo de Bellman-Ford

O método de Bellman-Ford é mostrado no Algoritmo 7.7. Ele efetua $n - 1$ relaxamentos de cada aresta no dígrafo. Ilustramos uma execução do algoritmo de Bellman-Ford na Figura 7.8.

Algoritmo BellmanFordShortestPaths($\vec{G}$,v):
  Entrada: um grafo ponderado e dirigido $\vec{G}$ e um vértice v de $\vec{G}$.
  Saída: um rótulo $D[u]$ para cada vértice u de $\vec{G}$ tal que $D[u]$ é o comprimento do caminho mínimo de v para u em $\vec{G}$, ou uma indicação de que $\vec{G}$ tem um ciclo de peso negativo
  $D[v] \leftarrow 0$
  para $u \neq v$ em $\vec{G}$ **faça**
    $D[u] \leftarrow +\infty$
  para $i \leftarrow 1$ até $n - 1$ **faça**
    para cada aresta dirigida (u,z) saindo de u **faça**
      {Realizar o *relaxamento* na aresta (u,z)}
      se $D[u] + w((u, z)) < D[z]$ **então**
        $D[z] \leftarrow D[u] + w((u, z))$
  se não há mais arestas com possíveis operações de relaxamento **então**
      retorne o rótulo $D[u]$ de cada vértice u
  senão
      retorne "$\vec{G}$ contém um ciclo negativo"

Algoritmo 7.7 O algoritmo de Bellman-Ford para caminhos mínimos a partir de um nodo do grafo, permitindo arestas com pesos negativos.

Lema 7.4: *se ao final da execução do Algoritmo 7.7 existir uma aresta* (u,z) *que pode ser relaxada (ou seja,* $D[u] + w((u,z)) < D[z]$*), então o dígrafo* $\vec{G}$ *contém um ciclo negativo. Senão,* $D[u] = d(v,u)$ *para cada vértice* u *em* $\vec{G}$.

Prova: para esta prova, vamos introduzir uma nova noção de distância em um dígrafo. Especificamente, seja $d_i(v, u)$ o comprimento de um caminho de v a u que é o mais curto dentre os caminhos de v a u com i arestas. Chamamos $d_i(v, u)$ de ***distância de i arestas*** de v até u. Afirmamos que, após a iteração i do laço principal do algoritmo de Bellman-Ford, temos $D[u] = d_i(v, u)$ para cada vértice em $\vec{G}$. Isto é verdadeiro antes que comecemos as iterações, pois $D[v] = 0 = d_0(v,v)$ e, para $u \neq v$, temos $D[u] = +\infty = d_0(v, u)$. Supondo que essa afirmação é verdadeira antes da iteração i, mostraremos que ela também será verdadeira após a iteração i. Nesta iteração, realizamos uma etapa de relaxamento para cada aresta no dígrafo. A distância de i arestas $d_i(v, u)$ de v a um vértice u é determinada de uma dentre duas formas: ou $d_i(v, u) = d_{i-1}(v, u)$ ou $d_i(v, u) = d_{i-1}(v, z) + w((z, u))$ para algum vértice z em $\vec{G}$. Pelo fato de fazermos um relaxamento para ***toda*** aresta de $\vec{G}$ na iteração i, se tivermos o primeiro caso, então após a iteração teremos $D[u] = d_{i-1}(v, u) = d_i(v, u)$, e se tivermos o segundo caso, teremos após a iteração $D[u] = D[z] + w((z, u)) = d_{i-1}(v, z) + w((z, u)) = d_i(v, u)$. Assim, se $D[u] = d_{i-1}(v, u)$ para cada vértice u antes da iteração i, então $D[u] = d_i(v, u)$ para cada vértice u após a iteração i.

Assim, após $n - 1$ iterações, $D[u] = d_{n-1}(v, u)$ para cada vértice u em $\vec{G}$. Observe que,se ainda houver uma aresta em $\vec{G}$ que pode ser relaxada então existe algum vértice u em $\vec{G}$ tal que a distância de n arestas de v até u é menor do que a distância de $n - 1$ arestas de v até u, ou seja, $d_n(v, u) < d_{n-1}(v, u)$. Mas existem apenas n vértices em $\vec{G}$; portanto, se existe um caminho mais curto com n arestas de v até u, então ele deve repetir algum vértice z de $\vec{G}$. Ou seja, $\vec{G}$ deve conter um ciclo. Além disso, como a distância de um vértice para si mesmo usando zero arestas é 0 (ou seja, $d_0(z, z) = 0$), este ciclo deve ter um peso total negativo. Assim, se existe uma aresta em $\vec{G}$ que ainda pode ser relaxada após executar o algoritmo de Bellman-Ford, então $\vec{G}$ contém um ciclo de peso negativo. Se,

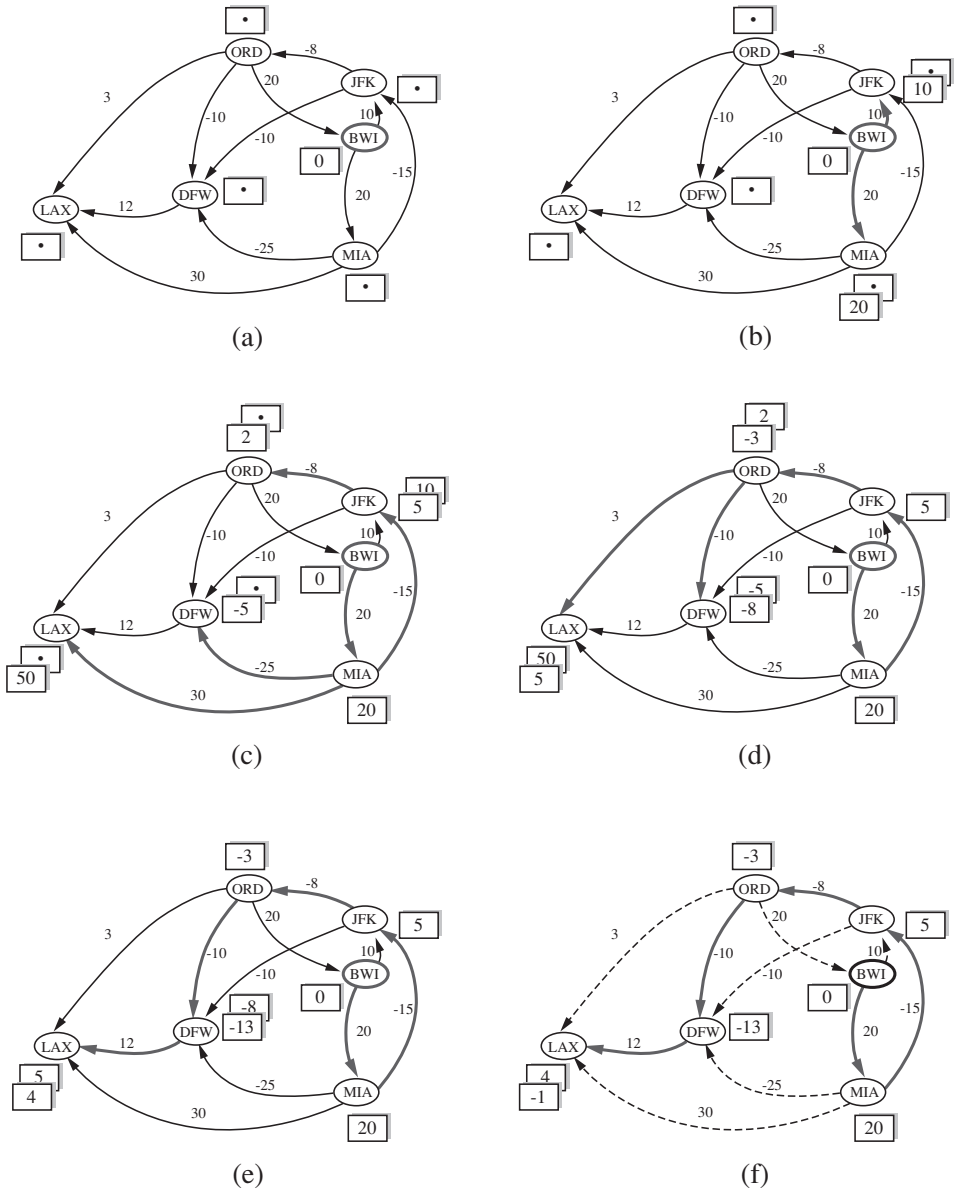


Figura 7.8 Ilustração da aplicação do algoritmo de Bellman-Ford. O vértice inicial é BWI. Uma caixa perto de cada vértice u armazena o valor de $D[u]$, com sombreado mostrando os valores revisados nos relaxamentos. As arestas mais espessas estão causando esses relaxamentos.

por outro lado, não há uma tal aresta, então $\vec{G}$ não contém ciclo de peso negativo. Além disso, neste caso, todo caminho mínimo entre dois vértices terá no máximo $n - 1$ arestas. Assim, para cada vértice u em $\vec{G}$, temos $D[u] = d_{n-1}(v, u) = d(v, u)$ ■

Portanto, o algoritmo de Bellman-Ford é correto e fornece uma maneira de determinar se um dígrafo contém um ciclo negativo. O tempo de execução do algoritmo de Bellman-Ford é de fácil análise. Realizamos o laço principal $n - 1$ vezes, e cada pas-

sagem pelo laço envolve tempo $O(1)$ para cada aresta em $\vec{G}$. Portanto, o tempo de execução deste algoritmo é $O(mn)$. Podemos resumir:

Teorema 7.5: *dado um grafo simples dirigido ponderado $\vec{G}$ com n vértices e m arestas e um vértice v de $\vec{G}$, o algoritmo de Bellman-Ford determina a distância de v a todos os outros vértices de $\vec{G}$ ou determina se $\vec{G}$ contém um ciclo negativo em tempo $O(mn)$.*

7.1.3 Caminhos mínimos em grafos acíclicos dirigidos

Como mencionado anteriormente, os algoritmos de Dijkstra e de Bellman-Ford operam sobre grafos dirigidos. Podemos, no entanto, resolver o problema de determinar os caminhos mínimos a partir de um nodo mais rapidamente se o dígrafo não tiver ciclos dirigidos, ou seja, se ele for um grafo acíclico dirigido ponderado (DAG*).

Recorde da Seção 6.4.4 que uma ordenação topológica de um DAG $\vec{G}$ é uma listagem de seus vértices $(v_1,v_2,...,v_n)$ de forma que se (v_i,v_j) for uma aresta em $\vec{G}$, então $i < j$. Também lembre que podemos usar o caminhamento em profundidade para determinar uma ordenação topológica dos n vértices em um DAG $\vec{G}$ com m arestas em tempo $O(n + m)$. Dada uma ordenação topológica de um DAG ponderado $\vec{G}$, podemos determinar todos os caminhos mínimos a partir de um vértice v em tempo $O(n + m)$.

Detalhes dos caminhos mínimos em um DAG

O método, dado no Algoritmo 7.9, envolve a visitação dos vértices de $\vec{G}$ de acordo com a ordenação topológica e relaxando as arestas de saída a cada visita.

Algoritmo DAGShortestPaths($\vec{G}$, s):
  Entrada: um grafo ponderado acíclico e dirigido $\vec{G}$ com n vértices e m arestas e um vértice s de $\vec{G}$.
  Saída: um rótulo $D[u]$ para cada vértice u de $\vec{G}$ tal que $D[u]$ é o comprimento do caminho mínimo de v para u em $\vec{G}$.
  Computa uma ordenação topológica $(v_1, v_2,...,v_n)$ para $\vec{G}$
  $D[s] \leftarrow 0$
  para $u \neq s$ em $\vec{G}$ **faça**
    $D[u] \leftarrow +\infty$
  para $i \leftarrow 1$ até $n - 1$ **faça**
    {Relaxa a aresta de saída de v_i}
    para cada aresta (v_i, u) saindo de v_i **faça**
      se $D[v_i] + w((v_i, u)) < D[u]$ **então**
        $D[u] \leftarrow D[v_i] + w((v_i, u))$
  Apresente os rótulos D como distâncias de s.

Algoritmo 7.9 Algoritmo de caminho mínimo para um grafo acíclico dirigido.

O tempo de execução do algoritmo de caminho mínimo para um DAG é de fácil análise. Assumindo que o dígrafo seja representado usando uma lista de adjacência, podemos processar cada vértice em tempo constante mais um tempo adicional proporcional ao número de suas arestas de saída. Já determinamos que encontrar uma ordenação topológica dos vértices em $\vec{G}$ leva tempo $O(n + m)$. Ilustramos este algoritmo na Figura 7.10.

* N. de R. Do inglês, *directed acyclic graph*.

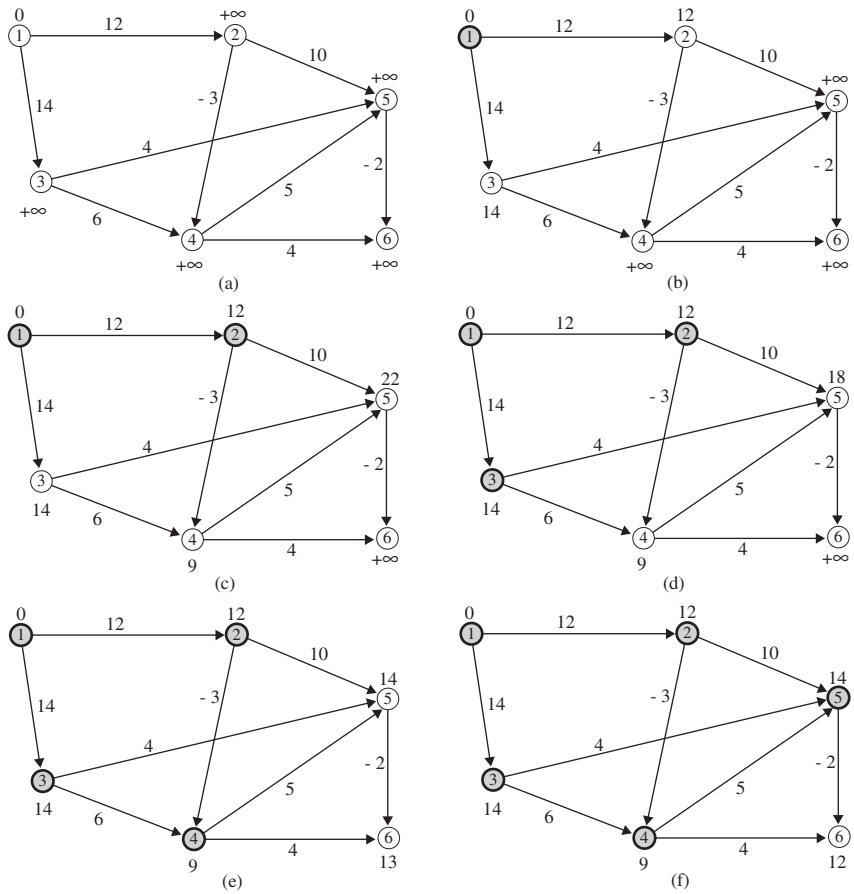


Figura 7.10 Algoritmo de caminho mínimo para um DAG.

Teorema 7.6: DAGShortestPaths *determina a distância desde um vértice inicial s até cada outro vértice em um grafo dirigido* $\vec{G}$ *com n vértices e m arestas em tempo* $O(n + m)$.

Prova: suponha que, para obtermos uma contradição, v_i seja o primeiro vértice na ordenação topológica tal que $D[v_i]$ não é a distância de s até v_i. Primeiramente, note que $D[v_i] < +\infty$, pois o valor inicial de D para qualquer vértice diferente de s é $+\infty$, e o valor do rótulo D somente pode ser diminuído se um caminho mais curto desde s for descoberto. Assim, se $D[v_j] = +\infty$, então v_j não pode ser alcançado a partir de s. Portanto v_i é atingível desde s, e existe um caminho mínimo de s a v_i. Seja v_k o penúltimo vértice no caminho mínimo de s a v_i. Como os vértices são numerados de acordo com uma ordenação topológica, sabemos que $k < i$. Assim, $D[v_k]$ está correto (podemos mesmo ter $v_k = s$). Porém quando v_k é processado relaxamos cada aresta de saída de v_k, incluindo a aresta do caminho mínimo de v_k a v_i. Assim, $D[v_i]$ recebe a distância de s até v_i, e isso contraria a definição de v_i. Portanto, não pode existir tal vértice. ■

7.2 Caminhos mínimos entre todos os vértices

Suponha que desejamos determinar os caminhos mínimos entre todos os pares de vértices em um grafo dirigido $\vec{G}$ com n vértices e m arestas. É claro que, se $\vec{G}$ não tiver arestas de peso negativo, poderíamos executar o algoritmo de Dijkstra iniciando a partir de cada vértice em $\vec{G}$. Essa abordagem levaria tempo $O(n(n + m) \log n)$, assumindo que $\vec{G}$ é representado através de uma lista de adjacência. No pior caso, esse tempo pode chegar a $O(n^3 \log n)$. De forma similar, se $\vec{G}$ não tiver ciclos negativos, podemos usar o algoritmo de Bellman-Ford iniciando a partir de cada vértice em $\vec{G}$. Esta abordagem levaria tempo $O(n^2m)$, que, no pior caso, pode chegar a $O(n^4)$. Nesta seção, consideraremos algoritmos para resolver o problema de encontrar os caminhos mínimos entre todos os vértices em tempo $O(n^3)$, mesmo se o dígrafo contiver arestas de peso negativo (mas sem ciclos negativos).

7.2.1 Um algoritmo de caminho mínimo com programação dinâmica

O primeiro algoritmo de caminho mínimo entre todos os pares que examinaremos é uma variação de um algoritmo mostrado anteriormente neste livro – o algoritmo de Floyd-Warshall para determinar o fechamento transitivo de um grafo dirigido (Algoritmo 6.16).

Seja $\vec{G}$ um grafo ponderado e dirigido. Numeramos os vértices de $\vec{G}$ arbitrariamente como $(v_1,v_2,...,v_n)$. Como em qualquer algoritmo de programação dinâmica (Seção 5.3), o ponto essencial do algoritmo é definir uma função de custo parametrizada que seja de fácil avaliação e que permita que determinemos a solução final. Neste caso, usamos a função de custo $D^k_{i,j}$, que é definida como a distância de v_i a v_j usando apenas vértices intermediários no conjunto $\{v_1,v_2,...,v_k\}$. Inicialmente,

$$D^0_{i,j} = \begin{cases} 0 & \text{se } i = j \\ w((v_i, v_j)) & \text{se } (v_i, v_j) \text{ for uma aresta em } \vec{G} \\ +\infty & \text{caso contrário.} \end{cases}$$

Dada a função de custo $D^k_{i,j}$ e seu valor inicial $D^0_{i,j}$, podemos definir seu valor para k arbitrário como

$$D^k_{i,j} = \min\{D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}\}.$$

Em outras palavras, o custo de ir de v_i a v_j usando vértices numerados de 1 a k é igual ao menor entre dois caminhos possíveis. O primeiro caminho é simplesmente o caminho mínimo de v_i a v_j usando os vértices numerados de 1 a $k - 1$. O segundo caminho é a soma dos custos do caminho mínimo de v_i a v_k usando os vértices numerados de 1 a $k - 1$ e o caminho mínimo de v_k a v_j usando os vértices numerados de 1 a $k - 1$. Além desses, não há outro caminho mínimo de v_i a v_j usando os vértices de $\{v_1,v_2,...,v_k\}$. Se houvesse um tal caminho e ele excluísse v_k, então ele violaria a definição de $D^{k-1}_{i,j}$, e se houvesse um tal caminho e ele incluísse v_k, então ele violaria a definição de $D^{k-1}_{i,j}$ ou $D^{k-1}_{k,j}$. De fato, repare que esse argumento se mantém mesmo que existam arestas de peso negativo em $\vec{G}$, desde que não existam ciclos negativos. No Algoritmo 7.11, mostramos como a definição desta função de custo permite construir uma solução eficiente para o problema do caminho mínimo entre todos os pares.

Algoritmo AllPairsShortestPaths($\vec{G}$):
  Entrada: um grafo ponderado dirigido $\vec{G}$ sem ciclos negativos.
  Saída: uma enumeração $v_1,v_2,...,v_n$ dos vértices de $\vec{G}$ e uma matriz D tal que $D[i,j]$ é a distância de v_i a v_j em $\vec{G}$.
  seja $v_1,v_2,...,v_n$ uma enumeração arbitrária dos vértices de $\vec{G}$
  para $i \leftarrow 1$ até n **faça**
    para $j \leftarrow 1$ até n **faça**
      se $i = j$ **então**
        $D^0 = [i, j] \leftarrow 0$
      se (v_i,v_j) é uma aresta em $\vec{G}$ **então**
        $D^0 = [i, j] \leftarrow w((v_i , v_j))$
      senão
        $D^0 = [i, i] \leftarrow +\infty$
  para $k \leftarrow 1$ até n **faça**
    para $i \leftarrow 1$ até n **faça**
      para $j \leftarrow 1$ até n **faça**
        $D^k[i, j] \leftarrow \min\{D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]\}$
  retorne matriz D^n

Algoritmo 7.11 Um algoritmo de programação dinâmica para determinar os caminhos mínimos entre todos os pares em um dígrafo sem ciclos negativos.

O tempo de execução para este algoritmo de programação dinâmica é evidentemente $O(n^3)$, e temos o teorema a seguir:

Teorema 7.7: *dado um grafo $\vec{G}$ simples, ponderado e dirigido com n vértices e sem ciclos negativos, o Algoritmo 7.11* (AllPairsShortestPaths) *determina as distâncias dos caminhos mínimos entre cada par de vértices em $\vec{G}$ em tempo $O(n^3)$.*

7.2.2 Determinação dos caminhos mínimos através da multiplicação de matrizes

Podemos visualizar o problema de determinar as distâncias dos caminhos mínimos para todos os pares de vértices em um grafo dirigido $\vec{G}$ como um problema matricial. Nesta subseção, descreveremos como resolver este problema em tempo $O(n^3)$ usando essa abordagem. Primeiramente, descreveremos como resolver o problema em tempo $O(n^4)$, e depois mostraremos como a solução pode ser melhorada chegando a tempo $O(n^3)$ se estudarmos o problema em maior profundidade. A abordagem através da multiplicação de matrizes é especialmente útil quando representamos grafos através de matrizes de adjacência.

A representação por matriz de adjacência ponderada

Vamos numerar os vértices de $\vec{G}$ como $v_0,v_1,...,v_{n-1}$, retornando à convenção de numerar os vértices iniciando em 0. Dada essa enumeração, existe uma forma natural de representar as arestas ponderadas de $\vec{G}$ em uma matriz, onde definimos $A[i,j]$ como abaixo:

$$A[i,j] = \begin{cases} 0 & \text{se } i = j \\ w((v_i, v_j)) & \text{se } (v_i, v_j) \text{ for uma aresta em } \vec{G} \\ +\infty & \text{caso contrário.} \end{cases}$$

(Note que essa é a mesma definição usada para a função de custo $D^0_{i,j}$ da subseção anterior.)

A multiplicação de matrizes e os caminhos mínimos

Em outras palavras, $A[i,j]$ armazena a distância no caminho mínimo de v_i a v_j usando uma aresta ou menos. Vamos usar a matriz A para definir outra matriz A^2, tal que $A^2[i,j]$ armazena a distância no caminho mínimo de v_i a v_j usando no máximo duas arestas. Um caminho de no máximo duas arestas pode ser vazio (nenhuma aresta) ou ele adiciona uma aresta extra a um caminho de zero aresta ou a um de uma aresta. Portanto, podemos definir $A^2[i,j]$ como

$$A^2[i,j] = \min_{l=0,1,\ldots,n-1} \{A[i,l] + A[l,j]\}$$

Assim, dada A, podemos determinar a matriz A^2 em tempo $O(n^3)$ usando um algoritmo muito similar ao algoritmo de multiplicação de matrizes.

De fato, podemos ver o cálculo de A^2 como uma multiplicação de matrizes na qual redefinimos os operadores "adição" e "multiplicação" (por exemplo, a linguagem de programação C++ permite esse tipo de redefinição). Se "adição" for redefinida como "min" e "multiplicação" for redefinida como "+", então podemos escrever $A^2[i,j]$ como uma verdadeira multiplicação de matrizes:

$$A^2[i,j] = \sum_{l=0,1,\ldots,n-1} \{A[i,l] \cdot A[l,j]\}.$$

De fato, essa abordagem é o motivo pelo qual escrevemos esta matriz como "A^2", pois ela é o quadrado da matriz A.

Vamos continuar definindo a matriz A^k, de forma que $A^k[i,j]$ seja a distância no caminho mínimo de v_i a v_j usando no máximo k arestas. Como um caminho de no máximo k arestas equivale a um caminho de no máximo $k-1$ arestas com uma possível aresta adicional, podemos definir A^k como

$$A^k[i,j] = \sum_{l=0,\ 1,\ldots,n-1} A^{k-1}[i,l] \cdot A[l,j],$$

continuando com a redefinição dos operadores "adição" como "min" e "multiplicação" como "+".

A observação crucial é que, se $\vec{G}$ não contém ciclos negativos, então A^{n-1} armazena a distância nos caminhos mínimos entre cada par de vértices em $\vec{G}$. Essa observação segue do fato de que qualquer caminho bem definido contém no máximo $n-1$ arestas. Se um caminho contiver mais de $n-1$ arestas, ele deve repetir algum vértice e portanto conter um ciclo. Mas um caminho mínimo nunca conterá um ciclo (a não ser que haja um ciclo negativo em $\vec{G}$). Portanto, para resolver o problema do caminho mínimo entre todos os pares, só o que precisamos é multiplicar A por si mesma $n-1$ vezes. Como cada uma destas multiplicações pode ser feita em tempo $O(n^3)$, esta abordagem imediatamente fornece o resultado a seguir.

Teorema 7.8: *dado um grafo $\vec{G}$, ponderado e dirigido com n vértices e que não contenha ciclos negativos, e a matriz de adjacência ponderada A de $\vec{G}$, o problema do caminho mínimo entre todos os pares em $\vec{G}$ pode ser resolvido calculando-se A^{n-1}, o que custa tempo $O(n^4)$.*

Na Seção 10.1.4, discutiremos um algoritmo de multiplicação para números que pode ser aplicado ao presente contexto de multiplicação de matrizes para calcular A^{n-1} em tempo $O(n^3\log(n))$. Mais do que isso, podemos calcular A^{n-1} em tempo $O(n^3)$

se tirarmos vantagem da estrutura presente no problema de achar o comprimento do caminho mínimo entre todos os pares.

Fechamento de uma matriz

Como observado acima, se $\vec{G}$ não contém ciclos negativos, então A^{n-1} codifica as distâncias mínimas entre os pares de vértices em $\vec{G}$. Um caminho mínimo bem definido não pode conter ciclos; portanto, um caminho mínimo limitado a conter no máximo $n - 1$ arestas deve ser um caminho mínimo verdadeiro. Da mesma forma, um caminho mínimo contendo no máximo n arestas é um caminho mínimo verdadeiro, como é um caminho mínimo contendo no máximo $n + 1$ arestas, $n + 2$ arestas e assim por diante. Portanto, se $\vec{G}$ não contém ciclos negativos, então

$$A^{n-1} = A^n = A^{n+1} = A^{n+2} = \cdots.$$

O ***fechamento*** de uma matriz A é definido como

$$A^* = \sum_{l=0}^{\infty} A^l,$$

se tal matriz existir. Se A é uma matriz de adjacência ponderada, então $A^*[i, j]$ é a soma de todos os possíveis caminhos de v_i a v_j. Em nosso caso, A é a matriz de adjacência ponderada para o grafo dirigido $\vec{G}$, e redefinimos "+" como "min". Assim, podemos escrever

$$A^* = \min_{i=0,\ldots,\infty} \{A^i\}.$$

Além disso, como estamos determinando as menores distâncias, as entradas em A^{i+1} nunca são maiores do que a entradas em A^i. Portanto, para a matriz de adjacência ponderada de um dígrafo $\vec{G}$ com n vértices sem ciclos negativos,

$$A^* = A^{n-1} = A^n = A^{n+1} = A^{n+2} = \cdots.$$

Ou seja, $A^*[i,j]$ armazena o comprimento do caminho mínimo de v_i a v_j.

Cálculo do fechamento de uma matriz de adjacência ponderada

Podemos calcular o fechamento A^* por divisão e conquista em tempo $O(n^3)$. Sem perda da generalidade, podemos assumir que n é uma potência de 2 (se não for, então adicionamos vértices extras a $\vec{G}$ sem ligá-los por arestas). Vamos dividir o conjunto V de vértices de $\vec{G}$ em dois conjuntos de igual tamanho $V_1 = \{v_0,\ldots,v_{n/2-1}\}$ e $V_2 = \{v_{n/2},\ldots, v_{n-1}\}$. Dada esta divisão, podemos dividir a matriz de adjacência A em quatro blocos, B, C, D e E, cada um com $n/2$ linhas e colunas, definidas como segue:

- B: pesos das arestas de V_1 para V_1;
- C: pesos das arestas de V_1 para V_2;
- D: pesos das arestas de V_2 para V_1;
- E: pesos das arestas de V_2 para V_2.

Ou seja,

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}.$$

Ilustramos esses quatro conjuntos de arestas na Figura 7.12.

De forma similar, podemos particionar A^* em quatro blocos, W, X, Y e Z, que são definidos de forma similar:

- W: pesos dos caminhos mínimos de V_1 para V_1;
- X: pesos dos caminhos mínimos de V_1 para V_2;
- Y: pesos dos caminhos mínimos de V_2 para V_1;
- Z: pesos dos caminhos mínimos de V_2 para V_2.

Ou seja,

$$A^* = \begin{pmatrix} W & X \\ Y & Z \end{pmatrix}.$$

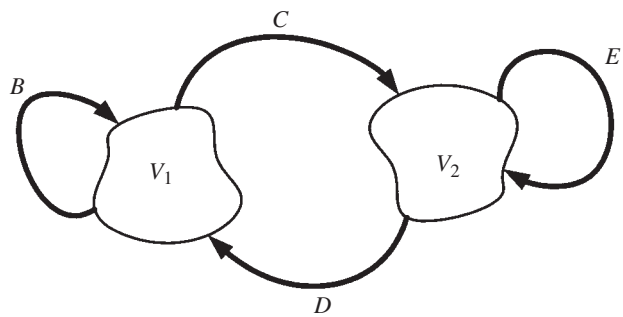


Figura 7.12 Uma ilustração dos quatro conjuntos de arestas usados para particionar a matriz de adjacência A no algoritmo de divisão e conquista para determinar A^*.

Equações das submatrizes

Pelas definições anteriores, podemos derivar equações simples para definir W, X, Y e Z diretamente das submatrizes B, C, D e E.

- $W = (B + C \cdot E^* \cdot D)^*$, pois os caminhos em W consistem no fechamento dos subcaminhos que ficam apenas em V_1 ou saem de V_1, vão até V_2, circulam em V_2 e depois retornam a V_1.
- $X = W \cdot C \cdot E^*$, pois os caminhos em X consistem no fechamento dos subcaminhos que iniciam e terminam em V_1 (com possíveis passagens por V_2), seguidos por uma ida a V_2 e o fechamento dos caminhos que ficam em V_2.
- $Y = E^* \cdot D \cdot W$, pois os caminhos em Y consistem do fechamento dos subcaminhos que ficam apenas em V_2, seguidos por uma ida a V_1 e o fechamento dos caminhos que iniciam e terminam em V_1 (com possíveis passagens por V_2).
- $Z = E^* + E^* \cdot D \cdot W \cdot C \cdot E^*$, pois os caminhos em Z consistem nos caminhos que ficam apenas em V_2 ou caminhos que saem de V_2, vão até V_1, circulam em V_1 (com possíveis passagens intermediárias por V_2) e depois retornam a V_2.

Dadas essas equações, é simples construir um algoritmo recursivo para calcular A^*. Neste algoritmo, dividimos A em blocos B, C, D e E como descritos acima. Em seguida calculamos recursivamente o fechamento E^*. Dado E^*, podemos calcular recursivamente o fechamento $(B + C \cdot E^* \cdot D)^*$, que é W.

Note que não são necessários outros fechamentos calculados recursivamente para determinar X, Y e Z. Assim, após um número constante de adições e multiplicações de matrizes, podemos determinar todos os blocos de A^*. Isso nos fornece o teorema a seguir.

Teorema 7.9: *dado um grafo $\vec{G}$, ponderado e dirigido com n vértices e não contendo ciclos negativos, e a matriz de adjacência ponderada A de $\vec{G}$, o problema do caminho mínimo entre todos os pares em $\vec{G}$ pode ser resolvido calculando-se A^*, o que custa tempo $O(n^3)$.*

Prova: já argumentamos que a determinação de A^* resolve o problema do comprimento dos caminhos mínimos entre todos os pares. Considere então o tempo de execução do algoritmo de divisão e conquista para determinar A^*, o fechamento da matriz de adjacência $A_{n\times n}$. Esse algoritmo consiste em duas chamadas recursivas para calcular o fechamento das submatrizes de tamanho $(n/2)\times(n/2)$ mais um número constante de adições e multiplicações de matrizes (usando "min" como "+" e "+" como " · "). Assumindo que usamos um algoritmo de tempo $O(n^3)$ para multiplicar as matrizes, podemos caracterizar o tempo de execução $T(n)$ para calcular A^* como

$$T(n)\begin{cases} b & \text{se } n=1 \\ 2T(n/2)+cn^3 & \text{se } n>1, \end{cases}$$

onde $b>0$ e $c>0$ são constantes. Portanto, pelo Teorema Mestre (5.6), podemos computar A^* em tempo $O(n^3)$. ■

7.3 Árvores de cobertura mínima

Suponha que desejamos conectar todos os computadores de um prédio de escritórios usando a menor quantidade possível de cabos, ou suponha que temos uma rede de computadores em que cada conexão entre dois roteadores tenha um custo de utilização e desejamos conectá-los com o menor custo possível. Podemos representar esses problemas usando um grafo ponderado G cujos vértices representam os computadores e cujas arestas representam todos os pares (u, v) de computadores possíveis onde o peso $w(u, v)$ da aresta (u, v) é igual ao comprimento dos cabos ou o custo necessário para ligar os computadores u e v. Em vez de determinar um caminho mínimo a partir de um dado vértice v, estamos interessados em encontrar uma árvore (livre) T que contenha todos os vértices de G e tenha o peso total mínimo. Os métodos para encontrar essas árvores são o foco desta seção.

Definição do problema

Dado um grafo não-dirigido ponderado G, desejamos encontrar uma árvore T que contenha todos os vértices de G e minimize a soma

$$w(T)=\sum_{e\in T} w(e).$$

Lembramos da Seção 6.1 que uma árvore como esta, que contém todos os vértices de um grafo conexo G, é chamada de ***árvore de cobertura***, e o problema de encontrar

uma árvore de cobertura T com a menor soma de pesos é conhecido como o problema da ***árvore de cobertura mínima*** (ou ***MST***[*]).

O desenvolvimento de algoritmos eficientes para o problema da árvore de cobertura mínima precede a noção moderna de Ciência da Computação. Nesta seção, discutiremos dois algoritmos para resolver o problema da MST. Esses algoritmos são aplicações clássicas do ***método guloso***, o qual, como discutimos na Seção 5.1, é baseado em escolher objetos para unirem-se a uma coleção crescente escolhendo iterativamente um objeto que minimiza uma dada função de custo.

O primeiro algoritmo que discutiremos é o algoritmo de Kruskal, que faz a MST "crescer" em grupos considerando as arestas na ordem dada por seus pesos. O segundo algoritmo que discutiremos é o algoritmo de Prim-Jarník, que faz a MST crescer a partir de um vértice raiz, de forma semelhante ao algoritmo de Dijkstra para determinação de caminhos mínimos. Concluímos esta seção discutindo um terceiro algoritmo, o de Barůvka, que aplica o método guloso de forma paralela.

Como na Seção 7.1.1, para simplificar a descrição dos algoritmos, assumimos que o grafo de entrada G é não-dirigido (ou seja, todas suas arestas são não-dirigidas) e simples (ou seja, os vértices não são ligados a si mesmos e não há arestas paralelas). Assim, denotamos as arestas de G como pares não-ordenados de vértices (u,z).

Um fato crucial sobre árvores de cobertura mínima

Antes de discutir os detalhes dos algoritmos, vamos examinar um fato crucial sobre as árvores de cobertura mínima que formam a base dos algoritmos. Todos os algoritmos para MST que discutiremos são baseados no método guloso, que, neste caso, depende crucialmente do fato a seguir (veja a Figura 7.13).

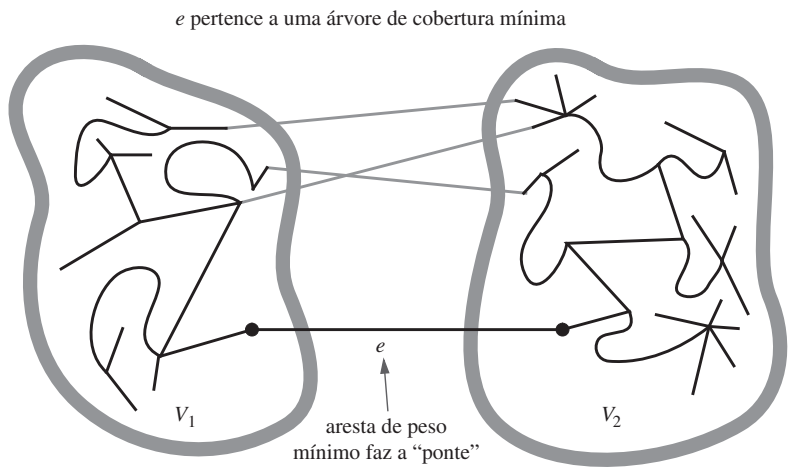


Figura 7.13 Uma ilustração do fato crucial sobre árvores de cobertura mínima.

Teorema 7.10: *seja G um grafo conexo ponderado e sejam V_1 e V_2 uma partição dos vértices de G em dois conjuntos disjuntos e não-vazios. Seja e uma aresta em G com*

[*] N. de R. Em inglês, *minimum spanning tree*.

peso mínimo dentre aquelas com um extremo em V_1 e outro em V_2. Existe uma árvore de cobertura mínima T que tem e como uma de suas arestas.

Prova: seja T uma árvore de cobertura mínima de G. Se T não contiver a aresta e, a adição de e a T criará um ciclo. Portanto, existe alguma aresta f deste ciclo que tem um extremo em V_1 e outro em V_2. Além disso, pela escolha de e, $w(e) \leq w(f)$. Se removermos f de $T \cup \{e\}$, obtemos uma árvore de cobertura mínima cujo peso total não é maior do que o anterior. Já que T é uma árvore de cobertura mínima, a nova árvore deve ser também uma árvore de cobertura mínima. ■

De fato, se os pesos de G são distintos, então a árvore de cobertura mínima é única; deixamos a prova deste fato menos crucial como exercício (C-7.5). Adicionalmente, note que o Teorema 7.10 permanece válido mesmo se o grafo G contiver arestas com pesos negativos ou ciclos com pesos negativos, diferentemente dos algoritmos que apresentamos para caminhos mínimos.

7.3.1 O algoritmo de Kruskal

O motivo do Teorema 7.10 ser tão importante é que ele pode ser usado como base para construir uma árvore de cobertura mínima. No algoritmo de Kruskal, ele é usado para construir a árvore de cobertura mínima em grupos. Inicialmente cada vértice é um grupo por si mesmo. O algoritmo então considera cada aresta, em ordem do menor para o maior peso. Se uma aresta e conecta dois grupos diferentes, então e é adicionada ao conjunto de arestas da árvore de cobertura mínima e os dois grupos ligados por e são unidos em um único grupo. Se, por outro lado, e conecta dois vértices que estão no mesmo grupo, então e é descartada. Uma vez que o algoritmo adiciona arestas suficientes para formar uma árvore de cobertura, ele termina e retorna a árvore como uma árvore de cobertura mínima.

Fornecemos o pseudocódigo para o algoritmo de Kruskal para a MST no Algoritmo 7.14 e mostramos o funcionamento deste algoritmo nas Figuras 7.15, 7.16 e 7.17.

Algoritmo KruskalMST(G):
  Entrada: um grafo simples, conexo e ponderado G com n vértices e m arestas.
  Saída: uma árvore de cobertura mínima T para G.
  para cada vértice v em G **faça**
    Defina um grupo elementar $C(v) \leftarrow \{v\}$.
  Inicialize uma fila de prioridade Q para conter todas as arestas em G,
  usando seus pesos como chaves.
  $T \leftarrow \emptyset$ {T irá conter as arestas da MST}
  enquanto T tem menos de $n - 1$ arestas **faça**
    $(u,v) \leftarrow Q$.removeMin()
    Seja $C(v)$ o grupo contendo v e $C(u)$ o grupo contendo u.
    se $C(v) \neq C(u)$ **então**
      Coloque aresta (v,u) em T.
      Unifique $C(v)$ e $C(u)$ em um grupo, isto é, a união de C(v) e C(u).
  retorne a árvore T

Algoritmo 7.14 Algoritmo de Kruskal para a MST.

Como mencionado antes, a correção do algoritmo de Kruskal resulta do fato crucial sobre árvores de cobertura mínima – Teorema 7.10. Cada vez que o algoritmo de Kruskal coloca uma aresta (u,v) na árvore de cobertura mínima T, podemos definir uma divisão do conjunto de vértices V (como na proposição) com V_1 sendo o grupo contendo v e V_2 contendo o resto dos vértices de V. Isso define claramente

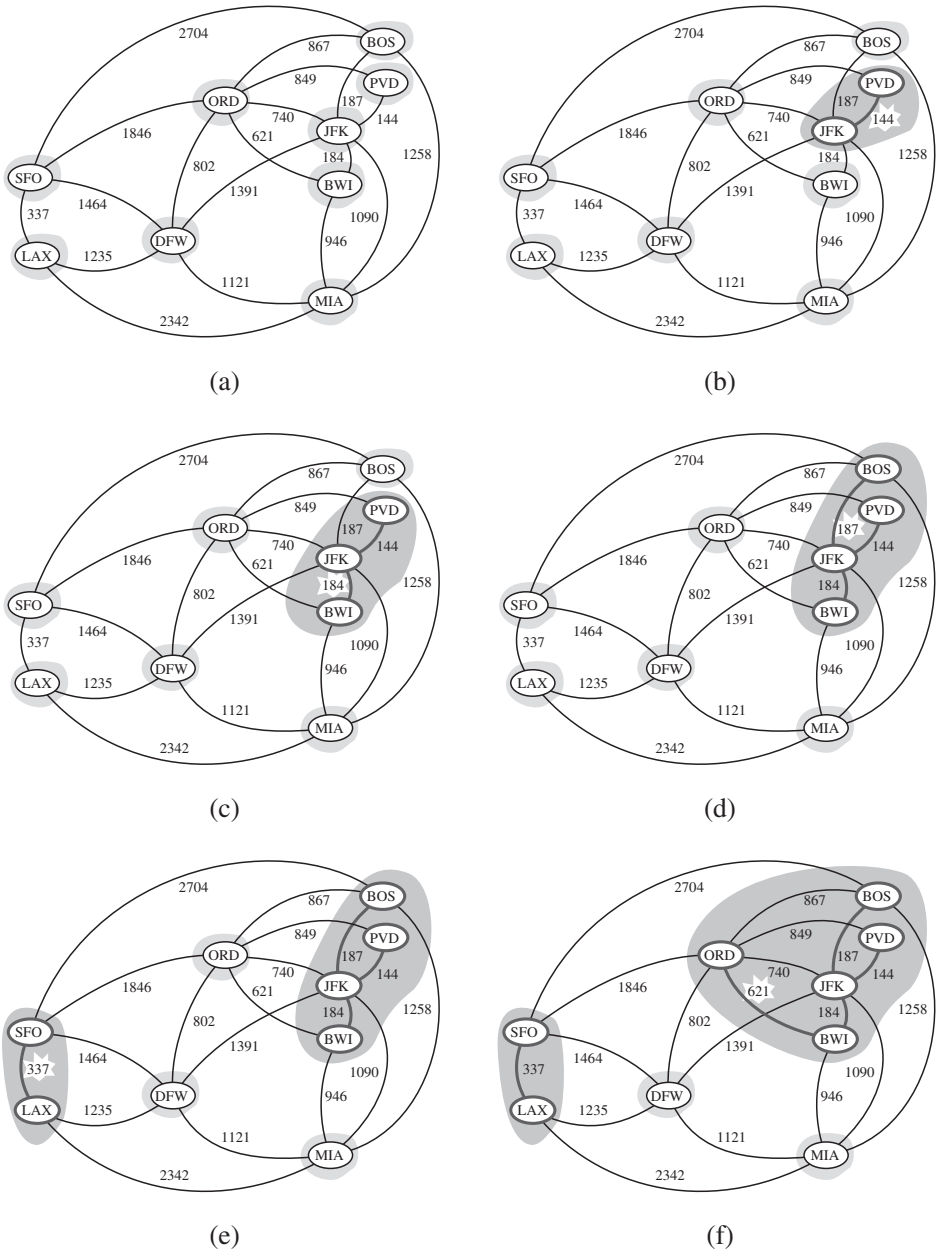


Figura 7.15 Exemplo da execução do algoritmo de Kruskal para MST em um grafo com pesos inteiros. Mostramos os grupos como regiões sombreadas e salientamos a aresta sendo examinada a cada iteração (continua na Figura 7.16).

uma divisão disjunta dos vértices de V e, mais importante, já que estamos retirando as arestas de Q na ordem de seus pesos, e deve ser uma aresta com peso mínimo com um vértice em V_1 e outro em V_2. Assim, o algoritmo de Kruskal sempre adiciona uma aresta válida à árvore de cobertura mínima.

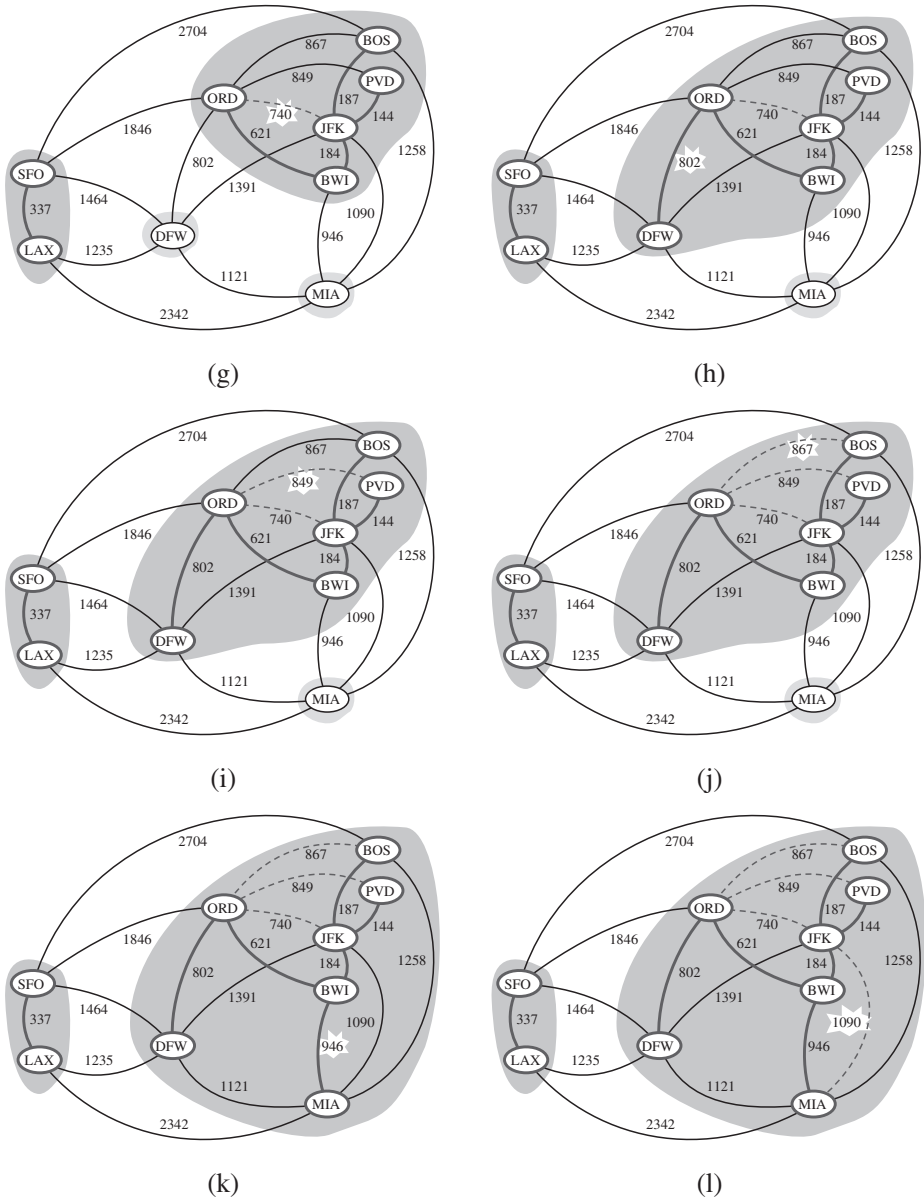


Figura 7.16 Exemplo da execução do algoritmo de Kruskal para MST (continuação). As arestas rejeitadas são mostradas em pontilhado (continua na Figura 7.17).

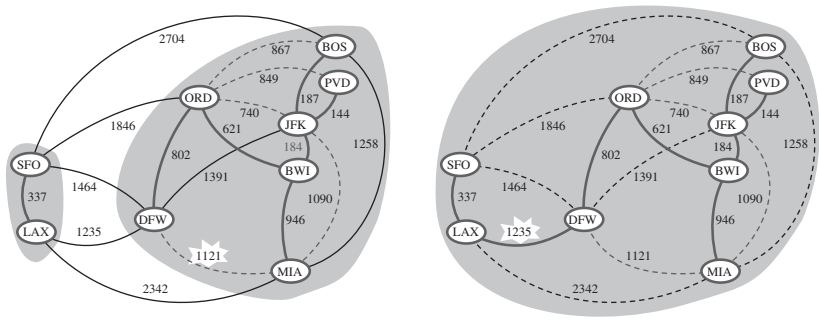


Figura 7.17 Exemplo da execução do algoritmo de Kruskal para MST (continuação das Figuras 7.15 e 7.16). A aresta considerada em (n) une os últimos dois grupos, o que conclui esta execução do algoritmo.

O tempo de execução do algoritmo de Kruskal

Nesta seção, analisaremos o tempo de execução do algoritmo de Kruskal. Denotamos com n e m o número de vértices e arestas do grafo G, respectivamente. Assumimos que os pesos das arestas podem ser comparados em tempo constante. Por causa do alto nível da descrição que fornecemos para o algoritmo de Kruskal no Algoritmo 7.14, analisar seu tempo de execução requer que tenhamos mais detalhes de sua implementação. Especificamente, devemos indicar as estruturas de dados usadas e a maneira como elas são implementadas.

Implementamos a fila de prioridade Q com um *heap*. Assim, podemos inicializar Q em tempo $O(m \log(m))$ através de inserções repetidas, ou em tempo $O(m)$ usando a construção *bottom-up* (veja a Seção 2.4.4). Adicionalmente, a cada iteração do laço **enquanto**, removemos uma aresta de peso mínimo em tempo $O(\log(m))$, que é na realidade $O(\log(n))$, pois G é simples.

Uma estratégia simples para os grupos

Podemos usar uma implementação com base em listas para partições (Seção 4.2.2) para os grupos. Ou seja, representamos cada grupo C com uma lista não-ordenada de vértices, que poderia, por exemplo, ser implementada com uma lista encadeada. Armazenamos com cada vértice v uma referência a seu grupo $C(v)$. Com essa representação, testar se $C(u) \neq C(v)$ leva tempo $O(1)$. Quando precisamos unir dois grupos $C(u)$ e $C(v)$, movemos os elementos do ***menor*** grupo para o maior e atualizamos as referências existentes nos elementos do menor grupo. Como podemos simplesmente adicionar os elementos do menor grupo ao final da lista de elementos do maior grupo, unificar os dois grupos leva tempo proporcional ao tamanho do menor grupo. Ou seja, unir os grupos $C(u)$ e $C(v)$ toma tempo $O(\min\{|C(u)|,|C(v)|\})$. Existem outros métodos mais eficientes para unificar os grupos (veja a Seção 4.2.2), mas esta abordagem será suficiente.

Lema 7.11: *considere uma execução do algoritmo de Kruskal em um grafo com n vértices onde grupos são representados com seqüências e com referências a grupos em cada vértice. O tempo total usado para unificar vértices é* $O(n \log(n))$.

Prova: observamos que cada vez que um vértice é movido a um novo grupo, o tamanho do grupo contendo o vértice no mínimo dobra. Seja $t(v)$ o número de vezes que o vértice v é movido a um novo grupo. Já que o tamanho máximo de um grupo é n,

$$t(v) \leq \log n.$$

O tempo total gasto unindo grupos no algoritmo de Kruskal pode ser obtido somando-se o trabalho feito a cada vértice, que é proporcional a

$$\sum_{v \in G} t(v) \leq n \log n.$$

■

Utilizando o Lema 7.11 e argumentos similares aos usados na análise do algoritmo de Dijkstra, concluímos que o tempo de execução total do algoritmo de Kruskal é $O((n + m)\log(n))$, que pode ser simplificado para $O(m \log(n))$, pois G é simples e conexo.

Teorema 7.12: *dado um grafo G simples, conexo e ponderado com n vértices e m arestas, o algoritmo de Kruskal constrói uma árvore de cobertura mínima para G em tempo $O(m \log(n))$.*

7.3.2 O algoritmo de Prim-Jarník

No algoritmo de Prim-Jarník, fazemos crescer uma árvore de cobertura mínima a partir de um único grupo iniciando com um vértice "raiz" v. A idéia principal é similar à do algoritmo de Dijkstra. Iniciamos com um vértice v, definindo a "nuvem" inicial de vértices C. A cada iteração, escolhemos uma aresta de peso mínimo $e = (v, u)$, conectando um vértice v da nuvem C a um vértice u fora de C. O vértice u é trazido para dentro da nuvem C, e o processo se repete até que uma árvore de cobertura seja formada. Novamente o fato crucial sobre árvores de cobertura mínima entra em ação, pois, se sempre escolhermos a aresta de menor peso unindo um vértice de C com um vértice fora de C, teremos certeza de estar sempre adicionando uma aresta válida à MST.

Aumentando uma única MST

Para implementar eficientemente esta abordagem, podemos usar outra idéia do algoritmo de Dijkstra. Mantemos um rótulo $D[u]$ para cada vértice u fora da nuvem C, armazenando o peso da melhor aresta corrente unindo u à nuvem C. Estes rótulos permi-

Algoritmo PrimJarnikMST(G):
  Entrada: um grafo simples, conexo e ponderado G com n vértices e m arestas.
  Saída: uma árvore de cobertura mínima T para G.
  Escolha qualquer vértice v de G
  $D[v] \leftarrow 0$
  para cada vértice $u \neq v$ **faça**
    $D[v] \leftarrow +\infty$
  Inicialize $T \leftarrow \emptyset$.
  Inicialize uma fila de prioridade Q com um item $((u,\text{null}), D[u])$ para cada vértice u onde (u, null) é o elemento e $D[u]$ é a chave.
  enquanto Q não estiver vazia **faça**
    $(u,e) \leftarrow Q$.removeMin()
    Coloque o vértice u e a aresta e em T
    para cada vértice z adjacente a u tal que z está em Q **faça**
      {faça o relaxamento na aresta (u,z)}
      se $w((u,z)) < D[z]$ **então**
        $D[z] \leftarrow w((u,z))$
        Altere para $(z,(u,z))$ o elemento do vértice z em Q.
        Altere para $D[z]$ a chave do vértice z em Q.
  retorne a árvore T

Algoritmo 7.18 O algoritmo de Prim-Jarník para a MST.

tem-nos reduzir o número de arestas que devemos considerar para decidir qual vértice deve ser unido à nuvem. Fornecemos o pseudocódigo no Algoritmo 7.18.

Analizando o algoritmo de Prim-Jarník

Sejam n e m o número de vértices e arestas do grafo de entrada G. A implementação do algoritmo de Prim-Jarník tem detalhes similares ao algoritmo de Dijkstra. Se implementarmos a fila de prioridade Q como um *heap* que suporta os métodos baseados em localizadores (veja a Seção 2.4.4), podemos extrair o vértice u a cada iteração em tempo $O(\log n)$.

Além disso, podemos atualizar cada valor de $D[z]$ em tempo $O(\log(n))$, o que é feito no máximo uma vez para cada aresta (u,z). Os outros passos de cada iteração podem ser implementados em tempo constante. Assim, o tempo de execução total do algoritmo é $O((n + m) \log(n))$, que é $O(m \log(n))$. Portanto, podemos sumarizar como segue:

Teorema 7.13: *dado um grafo G simples, conexo e ponderado com n vértices e m arestas, o algoritmo de Prim-Jarník constrói uma árvore de cobertura mínima para G em tempo* $O(m \log(n))$.

Ilustramos o algoritmo de Prim-Jarník nas Figuras 7.19 e 7.20.

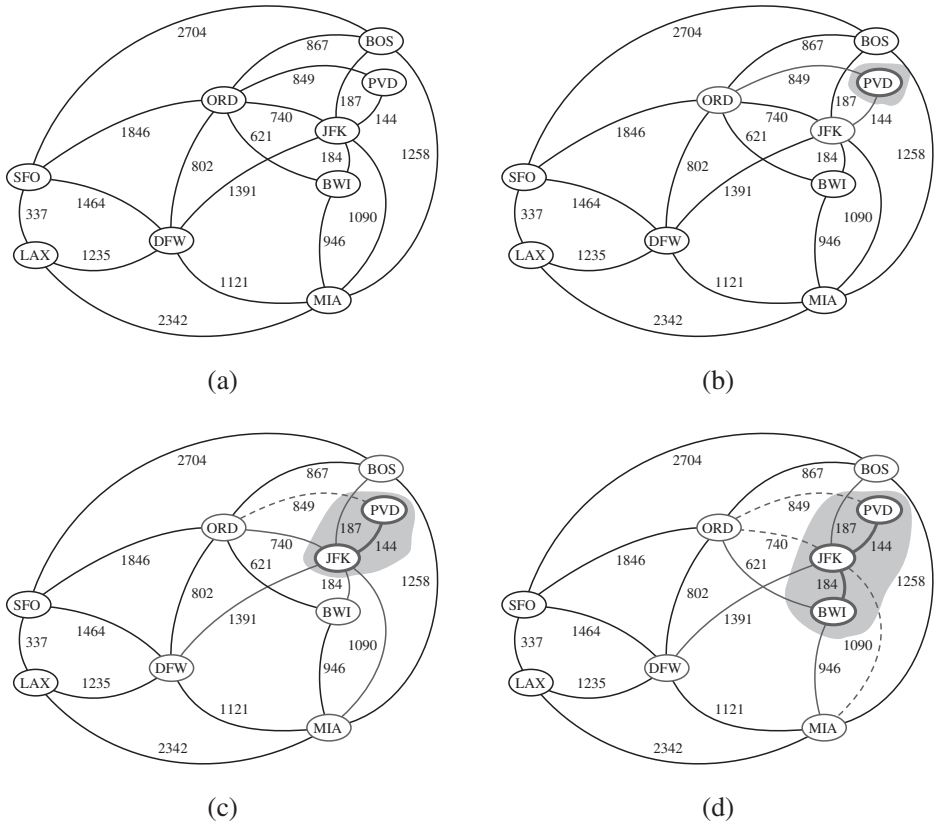


Figura 7.19 Ilustração do algoritmo de Prim-Jarník (continua na Figura 7.20).

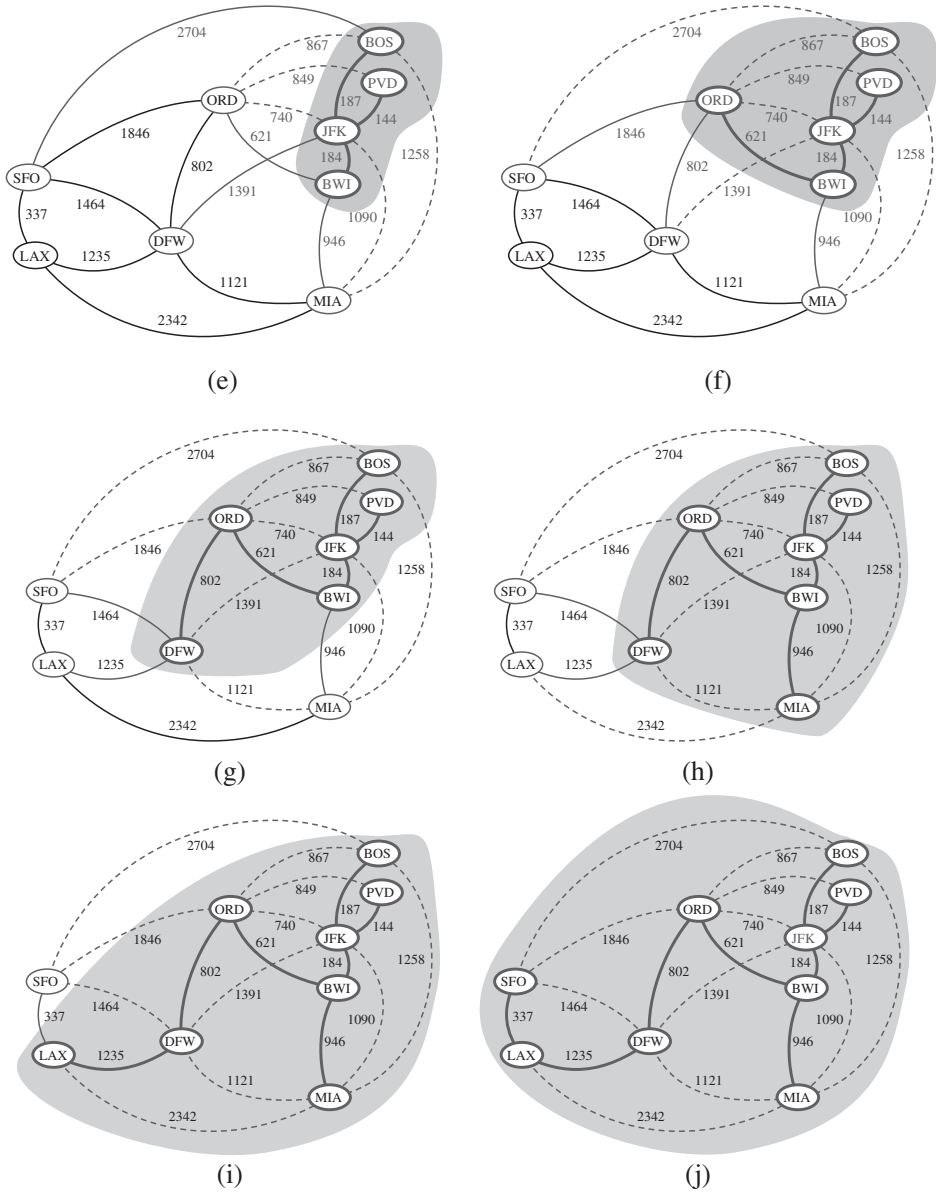


Figura 7.20 Ilustração do algoritmo de Prim-Jarník (continuação).

7.3.3 O algoritmo de Barůvka

Cada um dos algoritmos de MST que descrevemos atinge seu tempo de execução eficiente através de uma fila de prioridade Q que pode ser implementada usando-se um *heap* (ou outra estrutura de dados mais sofisticada). Esta escolha é bastante natural para algoritmos de determinação de árvores de cobertura mínima, pois os algoritmos envolvem o método guloso, e este precisa otimizar as prioridades entre os vértices a escolher. Pode ser um pouco surpreendente, como mostraremos nesta seção, que possa-

mos construir um algoritmo eficiente para a MST sem usar uma fila de prioridade. Pode ser ainda mais surpreendente quando observamos que esta simplificação origina-se no mais antigo algoritmo para determinação da árvore de cobertura mínima – o algoritmo de Barůvka. Apresentamos o pseudocódigo do algoritmo para determinação da árvore de cobertura mínima de Barůvka no Algoritmo 7.21, e ilustramos sua execução na Figura 7.22.

Algoritmo BarůvkaMST(G):
  Entrada: um grafo simples, conexo e ponderado $G = (V,E)$ com n vértices e m arestas.
  Saída: uma árvore de cobertura mínima T para G.
  Seja T um subgrafo de G inicialmente contendo apenas os vértices de V.
  enquanto T tem menos de $n - 1$ arestas {T ainda não é uma MST} **faça**
    para cada componente conexo C_i de T **faça**
      {Efetue o procedimento de adição de aresta na MST para o componente C_i }
      Encontre a aresta de menor peso $e = (v,u)$ em E com $v \in C_i$ e $u \notin C_i$.
      Adicione e a T (a não ser que e já esteja em T).
  retorne T

Algoritmo 7.21 Pseudocódigo para o algoritmo de Barůvka.

Implementação do algoritmo de Barůvka

Implementar o algoritmo de Barůvka é bastante simples e requer apenas que possamos efetuar as seguintes tarefas:

- manter a floresta T onde arestas serão inseridas, o que podemos fazer em tempo $O(1)$ usando uma lista de adjacência para T;
- percorrer a floresta T para identificar componentes conexos (grupos), o que podemos fazer em tempo $O(n)$ usando uma busca em profundidade em T;
- marcar vértices com o nome do grupo a que pertencem, o que podemos fazer com uma variável instanciada adicional para cada vértice;
- identificar a aresta de menor peso em E incidente a um grupo C_i, o que podemos fazer vasculhando as listas de adjacência em G para os vértices de C_i.

Assim como o algoritmo de Kruskal, o algoritmo de Barůvka constrói a MST fazendo aumentar os grupos de vértices em uma série de rodadas – não apenas um grupo, como no algoritmo de Prim-Jarník. Mas no algoritmo de Barůvka, os grupos aumentam aplicando-se o fato crucial sobre as árvores de cobertura mínima a cada grupo simultaneamente. Esta abordagem permite que muito mais arestas sejam adicionadas a cada rodada.

Por que o algoritmo está correto?

Em cada iteração do algoritmo de Barůvka, escolhemos a aresta de menor peso saindo de cada componente conexo C_i do conjunto corrente T de arestas da MST. Em cada caso, essa aresta é uma escolha válida, pois, se considerarmos uma divisão de V nos vértices em C_i e nos vértices fora de C_i, então a aresta e escolhida para C_i satisfaz a condição do fato crucial sobre as arvores de cobertura mínima (Teorema 7.10) e garante que e pertence à arvore de cobertura mínima.

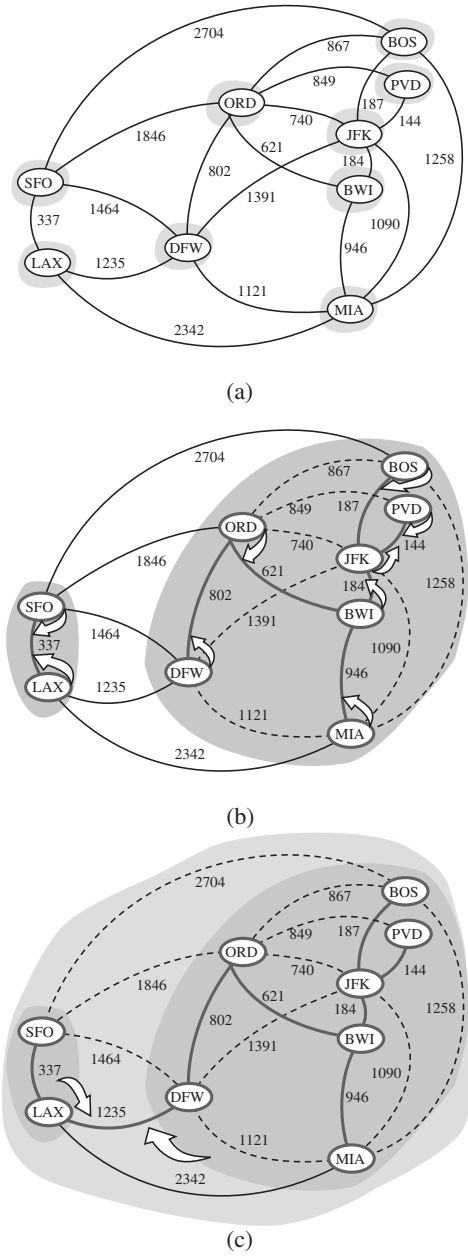


Figura 7.22 Exemplo da execução do algoritmo de Barůvka. Mostramos os grupos como regiões sombreadas. Salientamos a aresta escolhida para cada grupo com uma seta e desenhamos cada aresta da MST como uma linha mais espessa. As arestas que não farão parte da MST são mostradas em pontilhado.

Analisando o algoritmo de Barůvka

Vamos analisar o tempo de execução do algoritmo de Barůvka (Algoritmo 7.21). Podemos implementar cada rodada realizando as buscas para encontrar a aresta de menor

peso saindo de cada grupo com uma busca exaustiva nas listas de adjacência de cada vértice no grupo. Assim, o tempo de execução total gasto buscando a aresta de menor peso pode ser considerado $O(m)$, pois envolve examinar cada aresta (v, u) em G duas vezes: uma para v e uma para u (pois os vértices são rotulados com o número do grupo em que estão). As operações restantes no laço **enquanto** principal envolvem rotular novamente os vértices, o que toma tempo $O(n)$, e percorrer todas as arestas em T, o que toma tempo $O(n)$. Assim, cada rodada no algoritmo de Barůvka leva tempo $O(m)$ (pois $n \leq m$). Em cada rodada do algoritmo, escolhemos uma aresta saindo de cada grupo e unificamos cada novo componente conexo de T em um novo grupo. Assim, cada grupo antigo de T deve ser unificado com pelo menos um outro grupo antigo de T. Ou seja, em cada rodada do algoritmo de Barůvka, o número total de grupos é reduzido pela metade. Portanto, o número total de rodadas é $O(\log n)$, e o tempo de execução total do algoritmo de Barůvka é $O(m \log n)$. Podemos resumir:

Teorema 7.14: *o algoritmo de Barůvka determina uma árvore de cobertura mínima para um grafo conexo e ponderado G com n vértices e m arestas em tempo* $O(m \log n)$.

7.3.4 Comparação entre os algoritmos

Embora cada um dos algoritmos anteriores para determinar árvores de cobertura mínima tenha o mesmo tempo de execução no pior caso, cada um atinge este tempo de execução usando diferentes estruturas de dados e abordagens para construir a árvore de cobertura mínima.

No que tange a estruturas de dados auxiliares, o algoritmo de Kruskal usa uma fila de prioridade para armazenar arestas e uma coleção de conjuntos implementados como listas para armazenar os grupos. O algoritmo de Prim-Jarník usa apenas uma fila de prioridade para armazenar pares vértice-aresta. Assim, do ponto de vista de facilidade de programação, o algoritmo de Prim-Jarník é preferível. De fato, o algoritmo de Prim-Jarník é tão similar ao algoritmo de Dijkstra que uma implementação deste último pode ser convertida em uma implementação para o algoritmo de Prim-Jarník com pouco esforço. O algoritmo de Barůvka requer uma forma de representar os componentes conexos. Assim, do ponto de vista de facilidade de programação, os algoritmos de Prim-Jarník e Barůvka parecem mais vantajosos.

Em termos de fatores constantes, os três algoritmos são bastante similares, pois ambos têm constantes relativamente pequenas em seus tempos de execução assintóticos. O tempo de execução assintótico do algoritmo de Kruskal pode ser melhorado se as arestas forem ordenadas por seus pesos (usando uma estrutura de dados para partições como na Seção 4.2.2). O tempo de execução do algoritmo de Barůvka também pode ser alterado para $O(n^2)$ no pior caso com uma pequena alteração no algoritmo (que exploraremos no Exercício C-7.12). Assim, não há um claro vencedor entre os três algoritmos, embora o algoritmo de Barůvka seja o de mais fácil implementação.

7.4 Exemplo em Java: o algoritmo de Dijkstra

Nesta seção, apresentamos o código Java para realizar o algoritmo de Dijkstra (Algoritmo 7.2), assumindo que temos um grafo não-dirigido com pesos inteiros e positivos.

Expressamos nossa implementação do algoritmo de Dijkstra através de uma classe abstrata Dijkstra (Trecho de Código 7.23-7.25), que declara o método abstrato weight(e) para extrair o peso da aresta e. A classe Dijkstra deve ser estendida através de subclasses que implementam o método weight(e). Veja, por exemplo, a classe MyDijkstra no Trecho de Código 7.26.

```
/** O algoritmo de Dijkstra para o problema de caminho mínimo e origem única em
  * um grafo não-dirigido cujas arestas têm pesos inteiros não-negativos. Classes
  * estendendo esta classe abstrata devem definir o método weight(e),
  * que acessa o peso de uma aresta. */
public abstract class Dijkstra {
   /** Executa o algoritmo de Dijkstra. */
   public void execute(InspectableGraph g, Vertex source) {
      graph = g;
      dijkstraVisit(source);
   }
   /** Atributo para distância de vértices. */
   protected Object DIST = new Object();
   /** Ajusta a distância de um vértice. */
   protected void setDist(Vertex v, int d) {
      v.set(DIST, new Integer(d));
   }
   /** Determina a distância de um vértice até o vértice de origem. Este método
     * retorna o comprimento do caminho mínimo da origem até u após o método exe-
     * cute ter sido chamado. */
   public int getDist(Vertex u) {
      return ((Integer) u.get(DIST)).intValue();
   }
   /** Este método abstrato deve ser definido por subclasses
     * @return peso da aresta e. */
   protected abstract int weight(Edge e);
   /** Valor infinito. */
   public static final int INFINITE = Integer. MAX_VALUE;
   /** Grafo de entrada. */
   protected InspectableGraph graph;
   /** Fila de prioridade auxiliar. */
   protected PriorityQueue Q;
```

Trecho de Código 7.23 A classe Dijkstra implementando o algoritmo de Dijkstra (continua nos Trechos de Código 7.24 e 7.25).

O algoritmo é executado pelo método dijkstraVisit. Uma fila de prioridade Q suportando métodos baseados em localizadores (Seção 2.4.4) é usada. Inserimos um vértice u em Q com o método insert, que retorna o localizador de u em Q. Seguindo o padrão de localizadores, associamos a u seu localizador através do método setLoc e recuperamos o localizador de u através do método getLoc. Mudar o rótulo de um vértice

v para d no processo de relaxamento corresponde a chamar o método replaceKey(ℓ,d), onde ℓ é o localizador de z.

```
/** Execução do algoritmo de Dijkstra.
  * @param v vértice de origem. */
protected void dijkstraVisit (Vertex v) {
  // inicializa a fila de prioridade Q e armazena todos os vértices nela
  Q = new ArrayHeap(new IntegerComparator());
  para (VertexIterator vertices = graph.vertices(); vertices. hasNext();) {
    Vertex u = vertices. nextVertex();
    int u_dist;
    if (u==v)
      u_dist = 0;
    else
      u_dist = INFINITE;
    // setDist(u, u_Dist);
    Locator u_loc = Q.insert(new Integer(u_dist), u);
    setLoc(u, u_loc);
  }
  // faz crescer a nuvem, um vértice de cada vez
  while (!Q.isEmpty()){
    // remove de Q e insere na nuvem um vértice com distância mínima
    Locator u_loc = Q.min();
    Vertex u = getVertex(u_loc);
    int u_dist getDist(u_loc);
    Q.remove(u_loc); // remove u da fila de prioridade
    setDist(u, u_dist); // a distância de u é final
    destroyLoc(u); // remove o localizador associado a u
    if (u_dist == INFINITE)
      continue; // vértices não-atingíveis não são processados
    // examina todos os vizinhos de u e atualiza suas distâncias
    for (EdgeIterator edges = graph. incidentEdges(u); edges. hasNext();) {
      Edge e = edges. next Edge();
      Vertex z = graph.opposite(u, e);
      if (hasLoc(z))        {// verifica que z está em Q, ou seja, não está na nuvem
        int e_weight = weight(e);
        Locator z_loc = getLoc(z);
        int z_dist = getDist(z_loc);
        if ( u_dist + e_weight < z_dist        // relaxamento da aresta e = (u,z)
          Q.replaceKey(z_loc, new Integer(u_dist + e_weight));
      }
    }
  }
}
```

Trecho de Código 7.24 Método dijkstraVisit da classe Dijkstra.

```
/** Atributo para localizadores de vértices na fila de prioridade Q. */
protected Object LOC = new Object();
/** Verifica se há um localizador associado a um vértice. */
protected boollean hasLoc(Vertex v) {
   return v.has(LOC);
}
/** Acessa o localizador de um vértice em Q. */
protected Locator getLoc(Vertex v) {
   return (Locator) v.get(LOC);
}
/** Associa o localizador em Q com um vértice. */
protected void setLoc(Vertex v, Locator l) {
   v.set(LOC, l);
}
/** Remove o localizador associado com um vértice. */
protected void destroyLoc(Vertex v) {
   v.destroy(LOC);
}
/** Acessa o vértice associado com um localizador. */
protected Vertex getVertex(Locator l) {
   return (Vertex) l.element();
}
/** Acessa a distância de um vértice, dado seu localizador em Q. */
protected int getDist(Locator l) {
   return ((Integer) l.key()).intValue();
}
```

Trecho de Código 7.25 Métodos auxiliares da classe Dijkstra. Eles assumem que os vértices de grafo são decoráveis (continuação dos Algoritmos 7.23 e 7.24).

```
/** Uma especialização da classe Dijkstra que retira os pesos das arestas
  * das decorações. */
public class MyDijkstra extends Dijkstra {
   /** Atributo para pesos das arestas. */
   protected Object WEIGHT;
   /** Construtor que acerta o valor do peso. */
   public MyDijkstra (Object weight_attribute) {
      WEIGHT = weight_attribute;
   }
   /** O peso da aresta é armazenado no atributo WEIGHT da aresta. */
   public int weight(Edge e) {
      return ((Integer) e.get(WEIGHT)).intValue();
   }
}
```

Trecho de Código 7.26 Classe MyDijkstra, que estende Dijkstra e fornece uma implementação concreta do método weight(e).

7.5 Exercícios

Reforço

R-7.1 Desenhe um grafo simples, conexo e ponderado, com oito vértices e 16 arestas, cada um com um peso diferente. Identifique um vértice como "inicial" e mostre uma execução do algoritmo de Dijkstra neste grafo.

R-7.2 Demonstre como modificar o algoritmo de Dijkstra para o caso em que o grafo é dirigido e desejarmos determinar o caminho mínimo ***dirigido*** da origem aos outros vértices.

R-7.3 Demonstre como modificar o pseudocódigo do algoritmo de Dijkstra para não apenas determinar a distância de v para cada vértice em G, mas também para apresentar uma árvore T com raiz em v de tal forma que o caminho em T de v a um vértice u é o caminho mínimo em G de v até u.

R-7.4 Desenhe um grafo simples, dirigido e ponderado G com 10 vértices e 18 arestas, de forma que G contenha um ciclo de peso mínimo com pelo menos quatro arestas. Demonstre que o algoritmo de Bellman-Ford irá encontrar esse ciclo.

R-7.5 O algoritmo de programação dinâmica do Algoritmo 7.11 usa espaço $O(n^3)$. Descreva um versão desse algoritmo que use espaço $O(n^2)$.

R-7.6 O algoritmo de programação dinâmica do Algoritmo 7.11 encontra apenas as distâncias dos caminhos mínimos, e não os caminhos. Descreva uma versão desse algoritmo que apresente o conjunto de todos os caminhos mínimos entre cada par de vértices em um grafo dirigido. Seu algoritmo deve ter tempo $O(n^3)$.

R-7.7 Desenhe um grafo simples, conexo e dirigido G com oito vértices e 16 arestas, cada uma com um peso diferente. Demonstre a execução do algoritmo de Kruskal sobre esse grafo. (Observe que existe apenas uma árvore de cobertura mínima neste grafo.)

R-7.8 Repita o problema anterior para o algoritmo de Prim-Jarník.

R-7.9 Repita o problema anterior para o algoritmo de Barůvka.

R-7.10 Considere a implementação da fila de prioridade Q através de uma seqüência não-ordenada usada no algoritmo de Dijkstra. Qual o tempo de execução no melhor caso do algoritmo de Dijkstra em um grafo de n vértices?
Dica: considere o tamanho de Q cada vez que o elemento mínimo é retirado.

R-7.11 Descreva o significado das convenções gráficas usadas nas Figuras 7.3 e 7.4 ilustrando o algoritmo de Dijkstra. O que as setas significam? E as linhas mais grossas e pontilhadas?

R-7.12 Repita o Exercício R-7.11 para as Figuras 7.15 e 7.17 ilustrando o algoritmo de Kruskal.

R-7.13 Repita o Exercício R-7.11 para as Figuras 7.19 e 7.20 ilustrando o algoritmo de Prim-Jarník.

R-7.14 Repita o Exercício R-7.11 para a Figura 7.22 ilustrando o algoritmo de Barůvka.

Criatividade

C-7.1 Forneça um exemplo de um grafo simples de n vértices que faz com que o algoritmo de Dijkstra seja executado em tempo $\Omega(n^2 \log(n))$ quando é implementado com um *heap* como lista de prioridade.

C-7.2 Forneça um exemplo de um grafo ponderado e dirigido $\vec{G}$ com arestas de peso negativo mas sem um ciclo de peso negativo, de tal forma que o algoritmo de Dijkstra avalie incorretamente os caminhos mínimos desde um dado vértice v.

C-7.3 Considere a seguinte estratégia gulosa para achar o caminho mínimo do vértice *start* ao vértice *goal* em um dado grafo conexo:

1: Inicialize *path* com *start*.
2: Inicialize *VisitedVertices* com {*start*}.
3: Se *start* = *goal*, retorne *path* e termine. Em caso contrário, continue.
4: Encontre a aresta (*start*,*v*) de peso mínimo tal que *v* seja adjacente a *start* e *v* não esteja em *VisitedVertices*.
5: Adicione *v* a *path*.
6: Adicione *v* a *VisitedVertices*.
7: Faça *start* = *v* e vá para o passo 3.

Essa estratégia gulosa sempre encontra o caminho mínimo de *start* até *goal*? Explique por que ela funciona, ou forneça um contra-exemplo.

C-7.4★ Suponha que temos um grafo ponderado G com n vértices e m arestas, de forma que o peso de cada aresta é um inteiro entre 0 e n. Demonstre que podemos encontrar uma MST para G em tempo $O(n \log^*(n))$.

C-7.5 Demonstre que, se todos os pesos em um grafo conexo ponderado G são distintos, então existe exatamente uma árvore de cobertura mínima para G.

C-7.6 Forneça um algoritmo eficiente para determinar o caminho ***máximo*** dirigido de um vértice s a um vértice t em um dígrafo ponderado acíclico $\vec{G}$. Especifique a representação do grafo usada e quaisquer estruturas de dados usadas. Também analise a complexidade de seu algoritmo.

C-7.7 Suponha que você receba um diagrama de uma rede telefônica, que é um grafo G cujos vértices representam centrais e cujas arestas representam ligações entre essas centrais. As arestas são marcadas por suas capacidades de transmissão. A capacidade de transmissão de um caminho é a menor capacidade de transmissão de uma aresta do caminho. Forneça um algoritmo que, dado um diagrama da rede e duas centrais a e b, forneça a máxima capacidade de transmissão entre a e b.

C-7.8 A NASA deseja interligar n estações espalhadas nos Estados Unidos usando canais de comunicação. Cada par de estações tem uma capacidade de transmissão de mensagens diferente, que é conhecida de antemão. A NASA deseja escolher $n - 1$ canais (o mínimo possível), de tal forma que todas as estações estejam ligadas pelos canais de transmissão e a capacidade de transmissão total seja máxima. (Defina essa capacidade como a soma das capacidades individuais dos canais.) Forneça um algoritmo eficiente para

este problema e determine sua complexidade no pior caso. Considere o grafo ponderado $G = (V, E)$, onde V é o conjunto de estações e E é o conjunto de canais entre as estações. Defina o peso $w(e)$ de uma aresta $e \in$ E como a capacidade de transmissão do canal correspondente.

C-7.9 Suponha que você recebe uma *tabela de horários* que consista em:

- um conjunto A de n aeroportos e, para cada aeroporto $a \in A$, um tempo mínimo de conexão $c(a)$;
- um conjunto F de m vôos e para cada vôo $f \in F$:
  - um aeroporto de origem $a_1(f) \in A$,
  - um aeroporto de destino $a_2(f) \in A$,
  - hora de saída $t_1(f)$,
  - hora de chegada $t_2(f)$.

Descreva um algoritmo eficiente para o problema do escalonamento dos vôos. Neste problema, recebemos os aeroportos a e b, o tempo t e desejamos computar a seqüência de vôos que nos permite chegar o mais rápido possível em b saindo de a no tempo t ou mais tarde. O tempo mínimo de conexão nos aeroportos intermediários deve ser observado. Qual o tempo de execução de seu algoritmo em função de n e m?

C-7.10 Como recompensa por ter salvo o reino da Grã-Gracia do terrível monstro "Assintótico Exponencial", o rei dá a você a chance de uma grande recompensa. Atrás do castelo real, há um labirinto, e em cada passagem do labirinto, há uma sacola de moedas de ouro. A quantidade de ouro em cada sacola varia. Você terá a oportunidade de caminhar no labirinto recolhendo sacolas, entrando pela porta marcada como "Entrada" e saindo pela porta marcada como "Saída" (essas são portas diferentes). Quando estiver no labirinto, você não pode voltar em seu caminho. Cada corredor do labirinto tem uma seta pintada na parede e você só pode andar seguindo a direção das setas. Não existe maneira de fazer uma "volta" no labirinto. Você receberá um mapa do labirinto, incluindo as quantidades de ouro e as direções dos corredores. Descreva um algoritmo para ajudá-lo a recolher o máximo de ouro.

C-7.11 Suponha que recebemos um grafo dirigido $\vec{G}$ com n vértices e seja M a matriz de adjacência $n \times n$ correspondente a G.

a. Seja o produto de M consigo mesma (M^2) definido, para $1 \leq i, j \leq n$, como

$$M^2(i, j) = M(i, 1) \odot M(1, j) \oplus \cdots \oplus M(i, n) \odot M(n, j),$$

onde"$\oplus$" é o operador booleano **or** e "$\odot$" é o operador booleano **and**. Dada esta definição, o que é que $M^2(i, j) = 1$ informa sobre os vértices i e j? E se $M^2(i, j) = 0$?

b. Suponha que M^4 é o produto de M^2 consigo mesma. O que as entradas de M^4 representam? E as entradas de $M^5 = (M^4)(M)$? Em geral, que informação está contida na matriz M^p?

c. Suponha que $\vec{G}$ seja ponderado e assuma o seguinte:
   1: para $1 \leq i \leq n$, $M(i, i) = 0$.
   2: para $1 \leq i, j \leq n$, $M(i, j) = weight(i, j)$ se $(i, j) \in E$.

3: para $1 \leq i, j \leq n$, $M(i, j) = \infty$ se $(i, j) \notin E$.

Também defina M^2 para $1 \leq i,j \leq n$, como segue:

$$M^2(i, j) = \min\{M(i, 1) + M(1, j), ..., M(i, n) + M(n, j)\}.$$

Se $M^2(i, j) = k$, o que podemos concluir sobre a relação entre os vértices i e j?

C-7.12 Demonstre como modificar o algoritmo de Barůvka para que ele tenha um tempo de execução no pior caso $O(n^2)$.

Projetos

P-7.1 Implemente o algoritmo de Kruskal supondo que os pesos das arestas sejam inteiros.

P-7.2 Implemente o algoritmo de Prim-Jarník supondo que os pesos das arestas sejam inteiros.

P-7.3 Implemente o algoritmo de Barůvka supondo que os pesos das arestas sejam inteiros.

P-7.4 Realize uma comparação experimental entre dois dos algoritmos para MST discutidos neste capítulo (ou seja, escolha dois de Kruskal, Prim-Jarník e Barůvka). Desenvolva um conjunto de experimentos para testar os tempos de execução dos algoritmos usando grafos gerados aleatoriamente.

Notas

O primeiro algoritmo conhecido para a árvore de cobertura mínima é de Barůvka [22] e foi publicado em 1926. O algoritmo de Prim-Jarník foi primeiramente publicado em tcheco por Jarník [108] em 1930 e em inglês em 1957 por Prim [169]. Kruskal publicou seu algoritmo para a árvore de cobertura mínima em 1956 [127]. O leitor interessado em mais estudo sobre o problema da árvore de cobertura mínima pode consultar o artigo de Graham e Hell [89]. O algoritmo assintoticamente mais rápido até o momento para a árvore de cobertura mínima é um método randomizado de Karger, Klein e Tarjan [112], que tem tempo esperado $O(m)$.

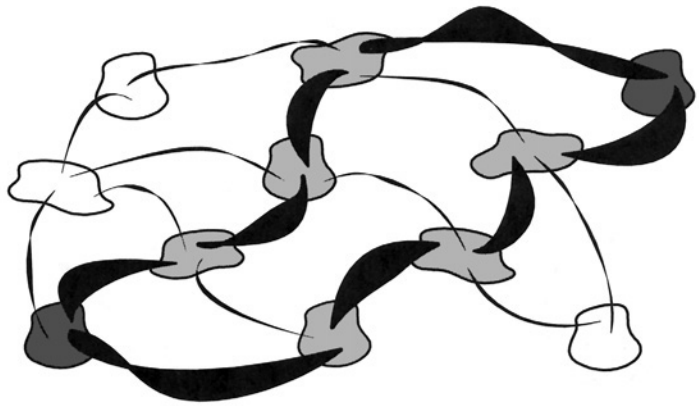
Dijkstra [60] publicou seu algoritmo para caminho mínimo com origem única em 1959. O algoritmo de Bellman-Ford é derivado de publicações separadas de Bellman [25] e Ford [71].

O leitor interessado em um estudo mais profundo sobre grafos pode consultar os livros de Ahuja, Magnanti e Orlin [9], Cormen, Leiserson e Rivest [55], Even [68], Gibbons [77], Mehlhorn [149], Tarjan [200] e o capítulo de van Leeuwen [205].

O tempo de execução para os algoritmos de Prim-Jarník e de Dijkstra pode ser melhorado até se tornar $O(n \log(n + m))$ implementando-se a fila de prioridade Q com uma de duas estruturas de dados mais sofisticadas, o "Fibonacci Heap" [72] ou o "Relaxed Heap" [61]. O leitor interessado nessas implementações deve ler os artigos que descrevem a implementação destas estruturas e como elas são aplicadas aos problemas de caminho mínimo e árvore de cobertura mínima.

Capítulo

8 Fluxo em redes e associação



8.1 Fluxos e cortes 382
8.1.1 Redes de Fluxo 382
8.1.2 Cortes 384
8.2 Fluxo máximo 386
8.2.1 Capacidade residual e aumento de caminhos 386
8.2.2 O algoritmo de Ford-Fulkerson 389
8.2.3 Análise do algoritmo de Ford-Fulkerson 391
8.2.4 O algoritmo de Edmonds-Karp 392
8.3 A associação máxima bipartida 395
8.3.1 Redução ao problema do fluxo máximo 395
8.4 Fluxo de custo mínimo 397
8.4.1 Ciclos aumentados 397
8.4.2 Caminhos mínimos sucessivos 398
8.4.3 Pesos modificados 399
8.5 Exemplo em Java: fluxo de custo mínimo 403
8.6 Exercícios 409

Um problema importante envolvendo grafos ponderados é o problema do ***fluxo máximo***. Neste problema, recebemos um grafo dirigido e ponderado *G* com cada aresta representando um "cano" que pode transportar algum produto, e o peso da aresta representando a quantidade máxima que ela pode transportar. O problema do fluxo máximo consiste em encontrar a forma de transportar o máximo do produto de um vértice *s* chamado de ***fonte*** até um vértice *t* chamado de ***destino***.

Exemplo 8.1: *considere uma parte da Internet modelada por um grafo dirigido G no qual cada vértice representa um computador, cada aresta (u, v) representa uma conexão de comunicação unidirecional de u para v, e o peso de cada aresta (u, v) representa a largura de banda do canal, ou seja, o maior número de bytes que podem ser mandados de u para v em um segundo. Se desejarmos uma transmissão de alta velocidade de um computador s em G para outro computador t em G, a maneira mais rápida é dividir a transmissão em pacotes e roteá-los através de G de acordo com o fluxo máximo* (*veja a Figura 8.1*).

O problema do fluxo máximo está relacionado ao problema de encontrar a melhor maneira de associar vértices de um tipo em um grafo com vértices de outro tipo. Portanto, também estudaremos o problema de associação, mostrando como o problema de fluxo máximo pode ser usado para resolvê-lo eficientemente.

Às vezes temos vários fluxos máximos diferentes. Embora todos sejam máximos em relação ao fluxo que produzem, esses fluxos podem ser diferentes em termos de custo. Assim, neste capítulo, também estudaremos métodos para computar os fluxos máximos que têm custo mínimo, quando existirem diferentes fluxos máximos e tivermos alguma forma de medir seus custos relativos. Concluiremos este capítulo com uma implementação Java de um algoritmo de fluxo de custo mínimo.

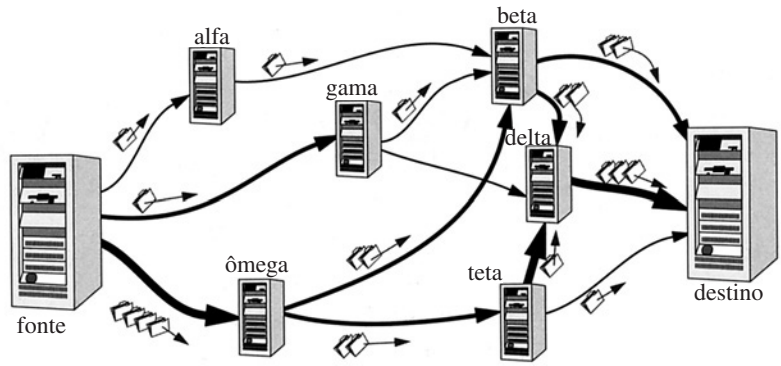


Figura 8.1 Um exemplo de fluxo em um grafo que representa uma rede de computadores com o comprimento de banda das conexões mais grossas de 4 MB/s, o comprimento de banda das conexões médias de 2 MB/s e o comprimento de banda das conexões finas de 1 MB/s. Indicamos a quantidade de dados mandados através de uma aresta por pastas, onde cada pasta corresponde a 1 MB/s passando pelo canal. Note que o fluxo total mandado da fonte ao destino (6 MB/s) não é máximo. De fato, um MB/s adicional pode ser mandado da fonte até gama, de gama para delta e de delta para o destino. Depois que este fluxo extra é adicionado, o fluxo total passa a ser máximo.

8.1 Fluxos e cortes

O exemplo anterior ilustra as regras que um fluxo válido deve seguir. Para afirmar precisamente quais são essas regras, vamos definir cuidadosamente o que chamamos de fluxo.

8.1.1 Redes de fluxo

Uma ***rede de fluxo*** N consiste no seguinte:

- um grafo conexo e dirigido G com pesos inteiros e não-negativos nas arestas, onde o peso de uma aresta e é chamado de ***capacidade*** $c(e)$ de e;
- dois vértices distintos s e t de G chamados de ***fonte*** e ***destino*** respectivamente, tais que s não tem arestas incidentes e t não tem arestas de saída.

Dado um tal grafo, o desafio é determinar a máxima quantidade de um produto que deve ser mandado de s a t sob a condição de que a capacidade de uma aresta determina o fluxo máximo que pode passar pela aresta (veja a Figura 8.2).

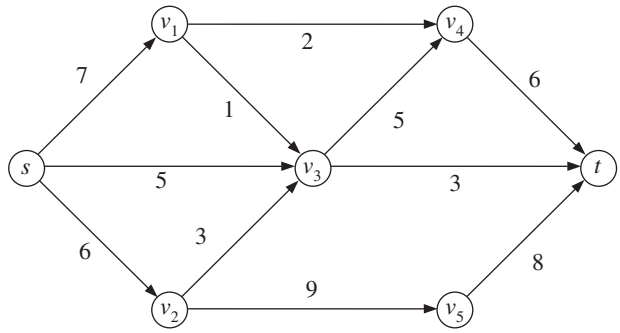


Figura 8.2 Uma rede de fluxo N. Cada aresta e de N é rotulada com sua capacidade $c(e)$.

Se desejamos que um produto seja levado de s a t, devemos ser mais precisos sobre o que queremos dizer com "fluxo". Um ***fluxo*** para a rede N é a atribuição de um valor inteiro $f(e)$ a cada aresta e de G que satisfaça às seguintes propriedades:

- para cada aresta e de G,

$$0 \leq f(e) \leq c(e) \text{ (\textbf{\textit{regra da capacidade}})};$$

- de cada vértice v de G diferente da fonte s e do destino t

$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e) \text{ (\textbf{\textit{regra da conservação}})},$$

  onde $E^-(v)$ e $E^+(v)$ denotam os conjuntos de arestas de chegada e de saída em v, respectivamente.

Em outras palavras, um fluxo deve satisfazer às restrições de capacidade das arestas e, para cada vértice v diferente de s e t, ter o fluxo total de saída de v igual ao fluxo de entrada em v. Cada uma das regras acima é respeitada, por exemplo, pelo fluxo ilustrado na Figura 8.3.

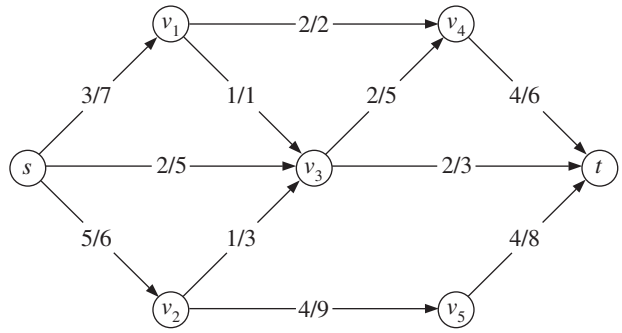


Figura 8.3 Um fluxo f (de valor $|f| = 10$) para a rede de fluxo N da Figura 8.2.

A quantidade $f(e)$ é chamada de ***fluxo*** da aresta e. O ***valor*** de um fluxo f, que denotamos por $|f|$, é igual ao fluxo total saindo da fonte s:

$$|f| = \sum_{e \in E^+(s)} f(e).$$

É fácil mostrar que o valor do fluxo é igual ao total de fluxo indo para o destino t (veja o Exercício R-8.1):

$$|f| = \sum_{e \in E^-(t)} f(e).$$

Ou seja, um fluxo especifica como um produto é levado de s através da rede N até o destino t. Um ***fluxo máximo*** para a rede de fluxo N é um fluxo com o maior valor entre todos os fluxos de N (veja a Figura 8.4). Já que um fluxo máximo usa a capacidade da rede da forma mais eficiente, estamos interessados em métodos para determinar fluxos máximos.

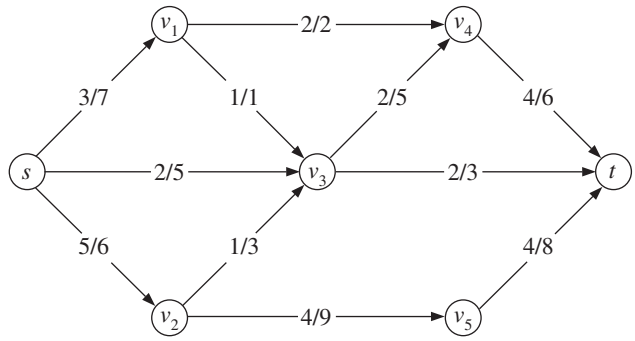


Figura 8.4 Um fluxo máximo f^* (de valor $|f^*| = 14$) para a rede de fluxo N da Figura 8.2.

8.1.2 Cortes

Fluxos são relacionados a outro conceito conhecido como corte. Intuitivamente, um corte é uma divisão dos vértices de uma rede de fluxo N em dois conjuntos, com s de um lado e t de outro. Formalmente, um ***corte*** de N é uma partição $\chi = (V_s, V_t)$ dos vértices de N de forma que $s \in V_s$ e $t \in V_t$. Uma aresta e de N com origem $u \in V_s$ e destino $v \in V_t$ é chamada de ***aresta de avanço*** do corte χ. Uma aresta com origem em V_t e destino em V_s é chamada de ***aresta de retorno***. Desejamos que um corte seja uma separação de s e t em N feita cortando-se arestas de N, indo arestas de avanço do lado de s para o lado de t e arestas de retorno para a direção oposta (veja a Figura 8.5).

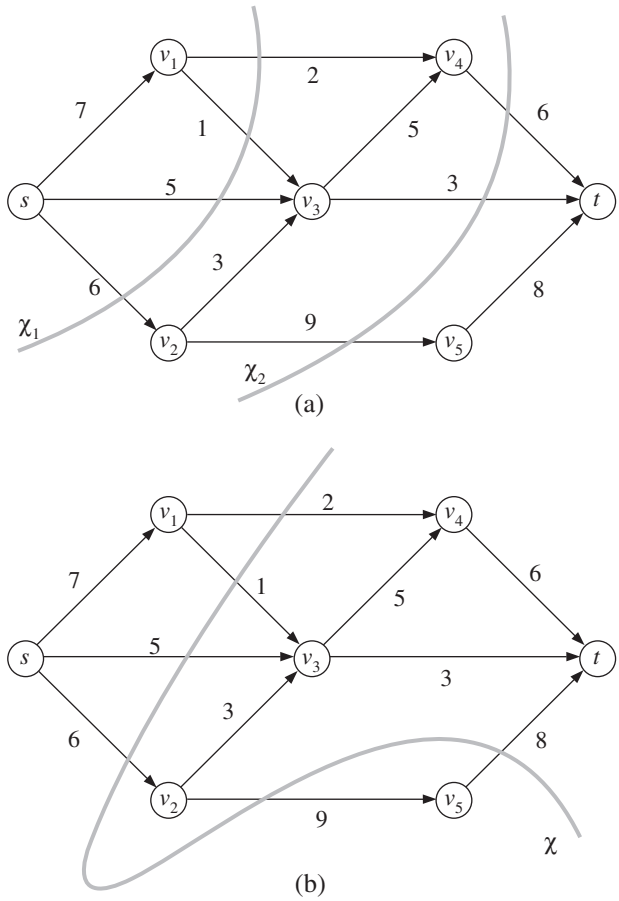


Figura 8.5 (a) Dois cortes χ_1 (à esquerda) e χ_2 (à direita) na rede de fluxo N da Figura 8.2. Esses cortes têm apenas arestas de avanço e suas capacidades são $c(\chi_1) = 14$ e $c(\chi_2) = 18$. O corte χ_1 é mínimo para N. (b) Um corte χ em N com arestas de avanço e de retorno. A capacidade $c(\chi)$ é 22.

Dado um fluxo f para N, o fluxo ***através do corte*** χ, denotado $f(\chi)$, é igual à soma dos fluxos das arestas de avanço de χ menos a soma dos fluxos das arestas de retorno de χ. Ou seja, $f(\chi)$ é o total de produto que passa do lado de s em χ ao lado de t em χ. O lema a seguir mostra uma propriedade interessante de $f(\chi)$.

Lema 8.2: *seja N uma rede de fluxo e f um fluxo em N. Para qualquer corte* χ *de N, o valor de f é igual ao fluxo através do corte* χ*, ou seja,* $|f| = f(\chi)$.

Prova: considere a soma

$$F = \sum_{v \in V_s} \left(\sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) \right)$$

Pela regra da conservação, para cada vértice v de V_s diferente de s temos que $\Sigma_{e \in E+(v)} f(e) - \Sigma_{e \in E-(v)} f(e) = 0$. Assim, $F = |f|$.

Por outro lado, para cada aresta e que não é uma aresta de avanço ou de retorno do corte χ, a soma F contém os termos $f(e)$ e $-f(e)$, que se cancelam, ou não contém nenhum dos dois, e, de qualquer forma, $F = f(\chi)$. ■

O teorema acima mostra que não importa onde cortamos uma rede de fluxo para separar s e t, pois o fluxo através do corte é igual ao fluxo para toda a rede. A ***capacidade*** do corte χ, denotada por $c(\chi)$, é a soma das capacidades das arestas de avanço de χ (note que não incluímos as arestas de retorno). O próximo lema mostra que uma capacidade de corte χ é um limite superior para qualquer fluxo através de χ.

Lema 8.3 *seja N uma rede de fluxo, e* χ *um corte de N. Dado qualquer fluxo f para N, o fluxo através do corte* χ *não ultrapassa a capacidade de* χ*, ou seja,* $f(\chi) \leq c(\chi)$.

Prova: denote por $E^+(\chi)$ as arestas de avanço de χ e por $E^-(\chi)$ as arestas de retorno de χ. Pela definição de $f(\chi)$, temos

$$f(\chi) = \sum_{e \in E^+(\chi)} f(e) - \sum_{e \in E^-(\chi)} f(e).$$

Retirando os termos não-positivos da soma acima, obtemos $f(\chi) \leq \Sigma_{e \in E^+(\chi)} f(e)$ Pela regra da capacidade, para cada aresta e, $f(e) \leq c(e)$. Assim, temos

$$f(\chi) = \sum_{e \in E^+(\chi)} c(e) = c(\chi).$$

■

Combinando os Lemas 8.2 e 8.3, obtemos o seguinte resultado importante relacionando fluxos e cortes:

Teorema 8.4: seja N uma rede de fluxo. Dado um fluxo f para N e qualquer corte χ de N, o valor de f não ultrapassa a capacidade de χ, ou seja, $|f| \leq c(\chi)$.

Em outras palavras, dado qualquer corte χ de uma rede de fluxo N, a capacidade de χ é um limite superior para qualquer fluxo em N. Esse limite superior é válido mesmo para um ***corte mínimo*** de N, que é um corte com mínima capacidade entre todos os cortes de N. No exemplo da Figura 8.5, χ_1 é um corte mínimo.

8.2 Fluxo máximo

O Teorema 8.4 implica que o valor de um fluxo máximo é limitado pela capacidade de um corte mínimo. Mostraremos nesta seção que essas duas quantidades são iguais. No processo, delinearemos uma abordagem para construir um fluxo máximo.

8.2.1 Capacidade residual e aumento de caminhos

Para provar que um dado fluxo f é máximo, precisamos de uma forma de provar que não existe mais nenhum fluxo adicional que possa ser "apertado" em f. Usando os conceitos relacionados de capacidade residual e aumento de caminhos, discutidos a seguir, podemos provar quando um fluxo f é máximo.

Capacidade residual

Seja N uma rede de fluxo, especificada por um grafo G, função de capacidade c, fonte s e destino t. Além disso, seja f um fluxo para N. Dada uma aresta e de G dirigida de um vértice u a um vértice v, a ***capacidade residual*** de u para v com respeito ao fluxo f, denotada por $\Delta_f(u, v)$, é definida como

$$\Delta_f(u, v) = c(e) - f(e),$$

e a capacidade residual de v para u é definida como

$$\Delta_f(v, u) = f(e).$$

Intuitivamente, a capacidade residual definida por um fluxo f é qualquer capacidade adicional que não tenha sido usada por f ao "levar" o fluxo de produto de s para t.

Seja π um caminho de s para t que pode passar por arestas de avanço ou de retorno, ou seja, podemos percorrer a aresta $e = (u, v)$ da origem u ao destino v ou do destino v para a origem u. Formalmente, uma ***aresta de avanço*** de π é uma aresta e de π tal que, indo de s para t pelo caminho π, a origem de e é encontrada antes do destino de e. Uma aresta de π que não é de avanço é dita ***aresta de retorno***. Vamos estender nossa definição de ***capacidade residual*** para uma aresta e em π percorrida de u para v, de forma que $\Delta_f(e) = \Delta_f(u, v)$. Em outras palavras,

$$\Delta_f(e) = \begin{cases} c(e) - f(e) & \text{se } e \text{ for uma aresta de avanço} \\ f(e) & \text{se } e \text{ for uma aresta de retorno.} \end{cases}$$

Ou seja, a capacidade residual de uma aresta e indo na direção de avanço é a capacidade adicional de e que f ainda deve consumir, mas a capacidade residual na direção oposta é o fluxo que f consumiu (e que poderia potencialmente "devolver", se isso permitir outro fluxo de valor mais alto).

Aumento de caminhos

A capacidade residual $\Delta_f(\pi)$ de um caminho π é a capacidade residual mínima de suas arestas. Ou seja,

$$\Delta_f(\pi) = \min_{e \in \pi} \Delta_f(e).$$

Esse valor é o total de fluxo adicional que podemos "colocar" pelo caminho π sem violar a condição de capacidade. Um ***aumento de caminho*** para o fluxo f é um caminho π da fonte s ao destino t com capacidade residual diferente de zero, ou seja, para cada aresta e em π,

- $f(e) < c(e)$ se e for uma aresta de avanço;
- $f(e) > 0$ se e for uma aresta de retorno.

Mostramos na Figura 8.6 um exemplo de um caminho aumentado.

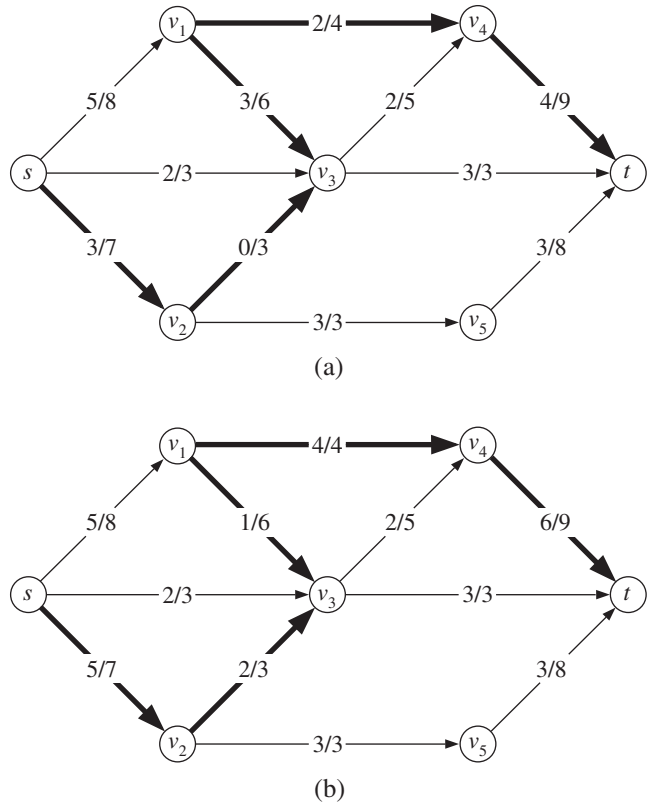


Figura 8.6 Exemplo de um caminho aumentado: (a) rede N, fluxo f e um caminho aumentado π representado com linhas grossas (a aresta (v_1, v_3) é de retorno); (b) fluxo f' obtido de f colocando $\Delta_f(\pi) = 2$ unidades de fluxo de s a t pelo caminho π.

Como mostrado no lema a seguir, podemos sempre adicionar a capacidade residual de um caminho aumentado a um fluxo existente e ter outro fluxo válido.

Lema 8.5: *seja π um caminho aumentado para o fluxo f na rede N. Existe um fluxo f' para N de valor $|f'| = |f| + \Delta_f(\pi)$.*

Prova: Computamos o fluxo f' modificando o fluxo das arestas em π:

$$f'(e) = \begin{cases} f(e) + \Delta_f(\pi) & \text{se } e \text{ for uma aresta de avanço} \\ f(e) - \Delta_f(\pi) & \text{se } e \text{ for uma aresta de retorno.} \end{cases}$$

Note que subtraímos $\Delta_f(\pi)$ se e for uma aresta de retorno, pois estamos retirando fluxo de e já tomado por f neste caso. De qualquer maneira, porque $\Delta_f(\pi) \geq 0$ é a menor capacidade residual de qualquer aresta em π, não violaremos a restrição de capacidade em uma aresta de avanço se adicionarmos $\Delta_f(\pi)$ nem teremos um fluxo menor do que zero em uma aresta de retorno se subtrairmos $\Delta_f(\pi)$. Assim, f' é um fluxo válido para N, e o valor de f' é $|f'| = |f| + \Delta_f(\pi)$. ■

Pelo Lema 8.5, a existência de um caminho aumentado π para um fluxo f implica que f não é máximo. Além disso, dado um caminho aumentado π, podemos modificar f para aumentar seu valor adicionando $\Delta_f(\pi)$ unidades de fluxo de s para t pelo caminho π, como mostrado na prova do Lema 8.5.

O que acontece se não houver um caminho aumentado para um fluxo f em uma rede N? Neste caso, temos que f é um fluxo máximo, como assegurado pelo lema a seguir.

Lema 8.6: *se uma rede N não tem um caminho aumentado com respeito a um fluxo f, então f é um fluxo máximo. Além disso, existem um corte χ de N tal que $|f| = c(\chi)$.*

Prova: seja f um fluxo para N, e suponha que não haja um caminho aumentado em N com respeito a f. Construímos a partir de f um corte $\chi = (V_s, V_t)$ colocando no conjunto V_s todos os vértices v que se ligam à fonte s através de arestas de capacidade residual não-nula. Tal caminho é chamado de caminho aumentado de s até v. O conjunto V_t contém os vértices restantes de N. Como não há caminho aumentado para o fluxo f, o destino t de N está em V_t. Assim, $\chi = (V_s, V_t)$ satisfaz a definição de um corte.

Pela definição de χ, cada aresta de avanço e de retorno do corte χ tem capacidade residual zero, ou seja,

$$f(e) = \begin{cases} c(e) & \text{se } e \text{ for uma aresta de avanço de } \chi \\ 0 & \text{se } e \text{ for uma aresta de retorno de } \chi. \end{cases}$$

Assim, a capacidade de χ é igual ao valor de f. Ou seja,

$$|f| = c(\chi)$$

Pelo Teorema 8.4, temos que f é um fluxo máximo. ■

Como conseqüência do Teorema 8.4 e do Lema 8.6, temos o seguinte resultado fundamental relacionando fluxos máximos e cortes mínimos.

Teorema 8.7 (Teorema Max-Flow, Min-Cut): *o valor de um fluxo máximo é igual à capacidade de um corte mínimo.*

8.2.2 O algoritmo de Ford-Fulkerson

Um algoritmo clássico, proposto por Ford e Fulkerson, determina o fluxo máximo em uma rede aplicando o método guloso à abordagem de caminhos aumentados usada para provar o Teorema Max-Flow, Min-Cut (Teorema 8.7).

A idéia principal do ***algoritmo de Ford-Fulkerson*** é aumentar o valor de um fluxo em estágios, onde, a cada estágio, uma certa quantidade de fluxo é colocada ao longo de um caminho aumentado da fonte ao destino. Inicialmente o fluxo de cada aresta é zero. A cada estágio, um caminho aumentado π é computado, e uma quantidade de fluxo igual à capacidade residual de π é adicionada no caminho π, como na prova do Lema 8.5. O algoritmo termina quando o fluxo corrente f não admite mais um caminho aumentado. O Lema 8.6 garante que neste caso f é um fluxo máximo.

Fornecemos uma descrição em pseudocódigo da solução de Ford-Fulkerson para o problema de encontrar um fluxo máximo no Algoritmo 8.7.

Algoritmo MaxFlowFordFulkerson(N):
  Entrada: rede de fluxo $N = (G, c, s, t)$
  Saída: um fluxo máximo f para N
 para cada aresta $e \in N$ **faça**
  $f(e) \leftarrow 0$
 stop $\leftarrow$ **falso**
 repita
  percorra G iniciando em s para encontrar um caminho aumentado para f
  se existe um caminho aumentado π **então**
   {Compute a capacidade residual $\Delta_f(\pi)$ de π }
   $\Delta \leftarrow +\infty$
   para cada aresta $e \in \pi$ **faça**
    se $\Delta_f(e) < \Delta$ **então**
     $\Delta \leftarrow \Delta_f(e)$
   {Coloque $\Delta = \Delta_f(\pi)$ unidades de fluxo pelo caminho π}
   para cada aresta $e \in \pi$ **faça**
    se e for uma aresta de avanço **então**
     $f(e) \leftarrow f(e) + \Delta$
    senão
     $f(e) \leftarrow f(e) - \Delta$ {e é uma aresta de retorno}
  senão
   stop $\leftarrow$ **verdadeiro** {f é um fluxo máximo}
 até *stop*

Algoritmo 8.7 O algoritmo de Ford-Fulkerson para determinar o fluxo máximo em uma rede.

Visualizamos o algoritmo de Ford-Fulkerson na Figura 8.8.

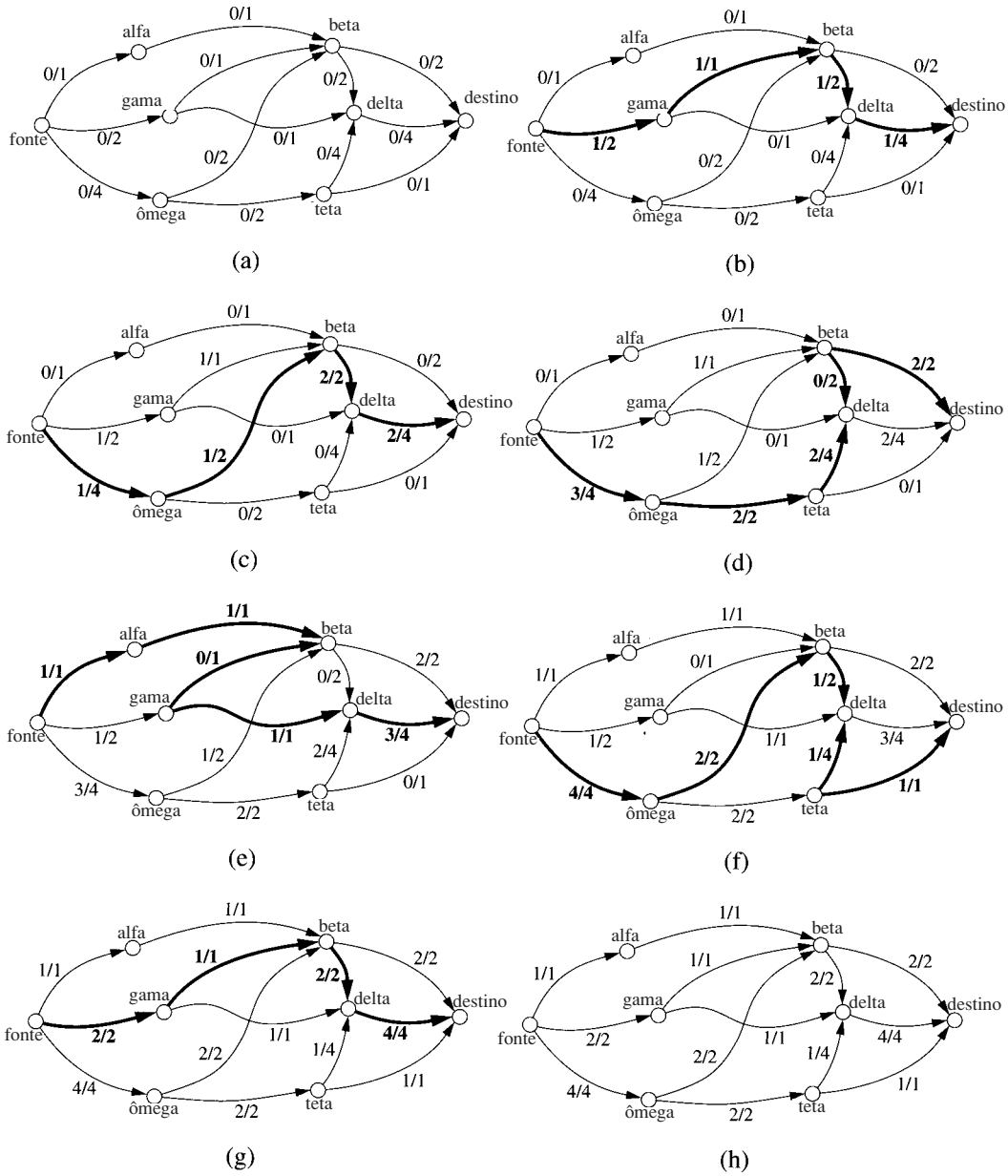


Figura 8.8 Execução do algoritmo de Ford-Fulkerson na rede de fluxo da Figura 8.1. Caminhos aumentados são mostrados com linhas grossas.

Detalhes de implementação

Existem detalhes importantes na implementação do algoritmo de Ford-Fulkerson que têm impacto na forma com que representamos o fluxo e como determinamos os caminhos aumentados. Representar um fluxo é bastante simples: podemos rotular cada

aresta da rede com um atributo representando o fluxo na aresta (seção 6.5). Para determinar um caminho aumentado, usamos um caminhamento especializado sobre o grafo G representando a rede. Este caminhamento é uma modificação simples de um caminhamento em profundidade (Seção 6.3.1) ou em largura (Seção 6.3.3), onde, em vez de considerar todas as arestas incidentes no vértice corrente v, consideramos somente as arestas a seguir:

- arestas de saída de v com fluxo menor do que a capacidade;
- arestas de chegada a v com fluxo não-nulo.

Alternativamente, a determinação de um caminho aumentado com respeito ao fluxo corrente f pode ser reduzida a um simples problema de determinação de caminho em um novo grafo dirigido R_f derivado de G. Os vértices de R_f são os mesmos vértices de G. Para cada par ordenado de vértices adjacentes u e v de G, adicionamos uma aresta dirigida de u para v se $\Delta_f(u, v) > 0$. O grafo R_f é chamado de ***grafo residual*** com respeito ao fluxo f. Um caminho aumentado com respeito ao fluxo f corresponde a um caminho dirigido de s a t no grafo residual R_f. Este caminho pode ser determinado por um caminhamento em profundidade de R_f iniciando na fonte s (veja as Seções 6.3 e 6.5).

8.2.3 Análise do algoritmo de Ford-Fulkerson

A análise do tempo de execução do algoritmo de Ford-Fulkerson é delicada. Isso acontece porque o algoritmo não especifica a forma exata de encontrar caminhos aumentados e, como veremos, a escolha do caminho aumentado tem um grande impacto no tempo de execução do algoritmo.

Sejam m e n os números de vértices e de arestas da rede de fluxo, respectivamente, e seja f^* um fluxo máximo. Como o grafo representando a rede é conexo, temos que $n \leq m + 1$. Note que cada vez que encontramos um caminho aumentado aumentamos o valor do fluxo em pelo menos uma unidade, pois as capacidades das arestas e fluxos são inteiros. Assim, o valor $|f^*|$ do fluxo máximo é um limite superior para o número de vezes em que o algoritmo procura um caminho aumentado. Note também que podemos encontrar um caminho aumentado através de um caminhamento sobre o grafo, como caminhamento em profundidade ou em largura, que custam tempo $O(m)$ (veja os Teoremas 6.13 e 6.19 e lembre que $n \leq m + 1$). Assim, podemos limitar o tempo de execução do algoritmo de Ford-Fulkerson como sendo no máximo $O(|f^*| m)$. Como ilustrado na Figura 8.9, este limite pode ser atingido para algumas escolhas de caminhos aumentados. Concluímos que o algoritmo de Ford-Fulkerson é um algoritmo de tempo pseudopolinomial (Seção 5.3.3), pois seu tempo de execução depende do tamanho da entrada e também do valor de um parâmetro numérico. Assim, o desempenho do algoritmo de Ford-Fulkerson pode ser baixo se $|f^*|$ for grande e os caminhos aumentados forem mal escolhidos.

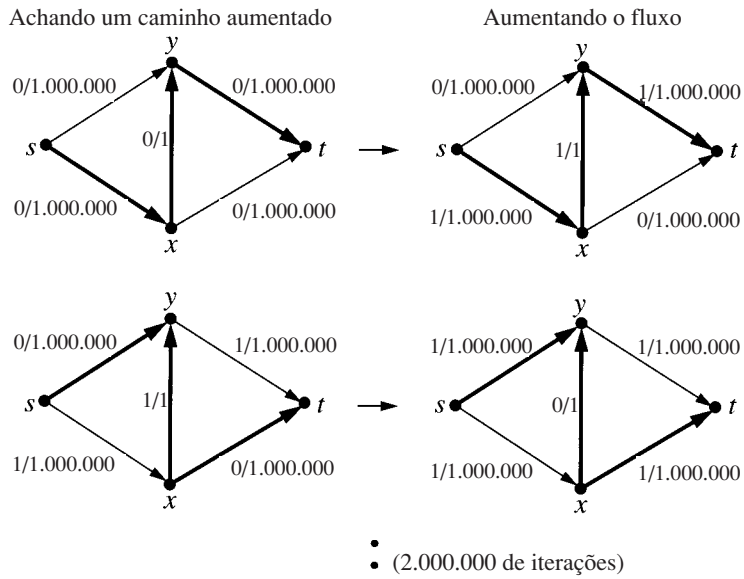


Figura 8.9 Exemplo de uma rede para a qual o algoritmo padrão de Ford-Fulkerson é lento. Se os caminhos aumentados escolhidos pelo algoritmo alternarem entre (s, x, y, t) e (s, y, x, t), então o algoritmo fará um total de 2 milhões de iterações, quando apenas duas iterações seriam suficientes.

8.2.4 O algoritmo de Edmonds-Karp

O ***algoritmo de Edmonds-Karp*** é uma variação do algoritmo de Ford-Fulkerson. Ele usa uma técnica simples para encontrar bons caminhos aumentados, o que resulta em um melhor tempo de execução. Esta técnica é baseada na noção de ser "mais guloso" em nossa aplicação do método guloso para o problema de fluxo máximo. Com isso, a cada iteração, escolhemos um caminho aumentado com o menor número de arestas, o que pode ser feito facilmente em tempo $O(m)$ com um algoritmo modificado de busca em largura. Mostraremos que, com esses ***aumentos de Edmonds-Karp***, o número de iterações não é maior do que mn, o que implica um tempo de execução $O(nm^2)$ para o algoritmo de Edmonds-Karp.

Começamos introduzindo a notação. Chamamos de ***comprimento*** de um caminho π o número de arestas em π. Seja f um fluxo para a rede N. Dado um vértice v, denotamos com $d_f(v)$ o comprimento mínimo de um caminho aumentado com respeito a f da fonte s ao vértice v, e chamamos essa quantidade de ***distância residual*** de v com respeito ao fluxo f.

A discussão a seguir mostra como a distância residual de cada vértice tem impacto sobre o tempo de execução do algoritmo de Edmonds-Karp.

Desempenho do algoritmo de Edmonds-Karp

Começamos nossa análise notando que a distância residual não decresce em uma seqüência de aumentos de Edmonds-Karp.

Lema 8.8: *seja g o fluxo obtido do fluxo f com um aumento sobre um caminho π de comprimento mínimo. Então, para cada vértice v,*

$$d_f(v) \leq d_g(v).$$

Prova: suponha que haja um vértice violando a desigualdade acima. Seja v este vértice com a menor distância residual com respeito a g. Ou seja,

$$d_f(v) > d_g(v) \tag{8.1}$$

e

$$d_g(v) \leq d_g(u), \text{ para cada } u \text{ tal que } d_f(u) > d_g(u). \tag{8.2}$$

Considere um caminho aumentado γ de comprimento mínimo de s até v considerando o fluxo g. Seja u o vértice imediatamente precedendo v em γ, e seja e a aresta de γ com pontos finais u e v (veja a Figura 8.10). Pela definição acima, temos

$$\Delta_g(u, v) > 0. \tag{8.3}$$

Assim, como u percede v imediatamente no caminho mínimo γ temos:

$$d_g(v) = d_g(u) + 1 \tag{8.4}$$

Finalmente, por (8.2) e (8.4), temos

$$d_f(u) \leq d_g(u). \tag{8.5}$$

Agora mostraremos que $\Delta_f(u, v) = 0$. De fato, se tivéssemos $\Delta_f(u, v) > 0$, poderíamos ir de u a v com um caminho aumentado considerando o fluxo f. Isso iria implicar

$$\begin{aligned} d_f(v) &\leq d_f(u)+1 \\ &\leq d_g(u)+1 && \text{por (8.5)} \\ &= d_g(v) && \text{por (8.4)} \end{aligned}$$

e contradizendo (8.1).

Como $\Delta_f(u, v) = 0$ e, por (8.3), $\Delta_g(u, v) > 0$, o caminho aumentado π que produz g a partir de f deve percorrer a aresta e de v a u (veja a Figura 8.10). Portanto,

$$\begin{aligned} d_f(v) &= d_f(u)-1 && \text{porque } \pi \text{ é um caminho mínimo} \\ &\leq d_g(u)-1 && \text{por (8.5)} \\ &\leq d_g(v)-2 && \text{por (8.4)} \\ &< d_g(v). \end{aligned}$$

Assim, obtemos uma contradição com (8.1), que completa a prova. ■

Intuitivamente, o Lema 8.8 implica que, a cada vez que fazemos um aumento de Edmonds-Karp, a distância residual de s a qualquer vértice v pode apenas aumentar ou se manter. Esse fato nos fornece o seguinte:

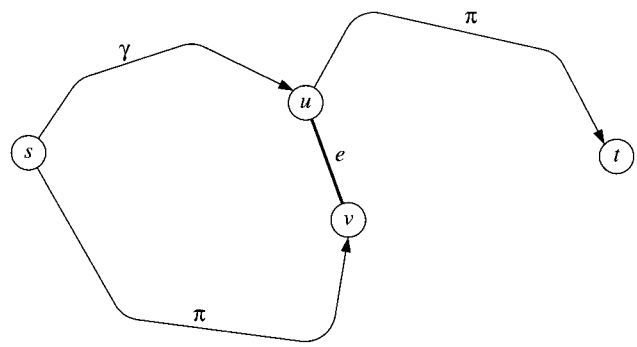


Figura 8.10 Ilustração da prova do Lema 8.8.

Lema 8.9: *quando executamos o algoritmo de Edmonds-Karp em uma rede com n vértices e m arestas, o número de aumentos de fluxo não é maior do que mn.*

Prova: seja f_i o fluxo na rede antes do i-ésimo aumento e seja π_i o caminho usado neste aumento. Dizemos que uma aresta e de π_i é um **gargalo** para π_i se a capacidade residual de e for igual à capacidade residual de π_i. É claro que cada caminho aumentado usado pelo algoritmo de Edmonds-Karp tem ao menos um gargalo.

Considere um par de vértices u e v unidos por uma aresta e e suponha que e seja um gargalo para dois caminhos aumentados π_i e π_k, com $i < k$, que percorrem e de u a v. As hipóteses acima implicam:

- $\Delta_{f_i}(u, v) > 0$;
- $\Delta_{f_{i+1}}(u, v) = 0$;
- $\Delta_{f_k}(u, v) > 0$.

Assim, deve existir um j-ésimo aumento intermediário com $i < j < k$, cujo caminho aumentado π_j percorre a aresta e de v para u. Portanto, obtemos

$$\begin{aligned} d_{f_j}(u) &= d_{f_j}(v) + 1 && \text{(porque } \pi_j \text{ é um caminho mínimo)} \\ &\geq d_{f_i}(v) + 1 && \text{(pelo Lema 8.8)} \\ &\geq d_{f_i}(u) + 2 && \text{(porque } \pi_i \text{ é um caminho mínimo).} \end{aligned}$$

Como a distância residual de um vértice é sempre menor do que o número de vértices n, cada aresta pode ser um gargalo no máximo n vezes durante a execução do algoritmo de Edmonds-Karp ($n/2$ vezes para para cada uma das duas direções em que ela pode ser percorrida por um caminho aumentado). Portanto, o número total de aumentos não é maior do que mn. ■

Como um único aumento de fluxo pode ser feito em tempo $O(m)$ usando um algoritmo modificado de busca em largura, podemos resumir a discussão acima como segue.

Teorema 8.10: *dada uma rede de fluxo com n vértices e m arestas, o algoritmo de Edmonds-Karp determina um fluxo máximo em tempo* $O(nm^2)$.

8.3 A associação máxima bipartida

Um problema que surge em uma série de aplicações é o problema da ***associação máxima bipartida***. Neste problema, temos um grafo não-dirigido e conexo G com as seguintes propriedades:

- os vértices de G são divididos em dois conjuntos X e Y;
- cada aresta de G tem um ponto final em X e um ponto final em Y.

Um grafo assim é chamado de ***grafo bipartido***. Uma ***associação*** em G é um conjunto de arestas que não têm pontos finais em comum — um conjunto assim cria "pares" de vértices em X e vértices em Y de forma que cada vértice tem no máximo um "associado" no outro conjunto. O problema da máxima associação bipartida é encontrar uma associação com o maior número de arestas (dentre todas as associações).

Exemplo 8.11: *seja G um grafo bipartido onde o conjunto X representa um grupo de rapazes e o conjunto Y representa um conjunto de garotas, todos juntos em uma festa. Suponha que exista uma aresta ligando x em X e y em Y se x e y querem dançar juntos. Uma associação máxima em G corresponde ao maior conjunto de pares compatíveis de rapazes e garotas que podem estar dançando ao mesmo tempo.*

Exemplo 8.12: *seja G um grafo bipartido onde o conjunto X representa disciplinas na universidade e o conjunto Y representa um conjunto de salas de aula. Suponha que exista uma aresta ligando x em X e y em Y se, de acordo com as matrículas e necessidades audiovisuais, a disciplina x puder ser ministrada na sala y. Uma associação máxima em G corresponde ao maior conjunto de disciplinas que podem ser ministradas simultaneamente sem conflito de espaço.*

Esses dois exemplos fornecem uma pequena amostra dos tipos de aplicações que o problema da associação máxima bipartida pode resolver. Felizmente, existe uma maneira simples de resolver o problema da associação máxima bipartida.

8.3.1 Redução ao problema do fluxo máximo

Seja G um grafo bipartido cujos vértices são divididos em conjuntos X e Y. Criamos uma rede de fluxo H tal que o fluxo máximo em H pode ser convertido imediatamente em uma associação máxima em G:

- Começamos incluindo todos os vértices de G em H, mais um novo vértice fonte s e um novo vértice destino t.
- Adicionamos cada aresta de G em H, mas alteramos a direção das arestas de forma que elas saiam do conjunto X e cheguem ao conjunto Y. Além disso, inserimos uma aresta dirigida de s para cada vértice em X e uma aresta dirigida de cada vértice em Y para t. Finalmente, associamos a cada aresta em H a capacidade de 1.

Dado um fluxo f para H, usamos f para definir um conjunto M de arestas de G usando a regra de que uma aresta e está em M sempre que $f(e) = 1$ (veja a Figura 8.11). Mostraremos agora que o conjunto M é uma associação. Como as capacidades em H são todas 1, o fluxo através de cada aresta de H é 0 ou 1. Além disso, como cada vértice x em X tem exatamente uma aresta de chegada, a regra da conservação implica que

no máximo uma das arestas de saída de x tenha fluxo não-negativo. Da mesma forma, cada vértice y em Y tem exatamente uma aresta de saída e, por isso, somente uma aresta de entrada em y tem fluxo não-negativo. Assim, cada vértice em X vai ser associado por M com no máximo um vértice em Y, ou seja, o conjunto M é uma associação. Também podemos verificar que o tamanho de M é igual a $|f|$, o valor do fluxo f.

A transformação inversa também pode ser definida: dada uma associação M em um grafo G, podemos usar M para definir um fluxo f para H usando as regras a seguir:

- Para cada aresta e em H que também está em G, $f(e) = 1$ se $e \in M$ e $f(e) = 0$ em caso contrário.
- Para cada aresta e em H incidente em s ou t, $f(e) = 1$ se v é um ponto final de alguma aresta de M e $f(e) = 0$ em caso contrário, onde v denota o outro ponto final de e.

É fácil verificar que f é um fluxo em H e o valor de f é igual ao tamanho de M.

Portanto, qualquer algoritmo de fluxo máximo pode ser usado para resolver o problema da associação máxima bipartida em um grafo G com n vértices e m arestas. Ou seja:

1. Construímos a rede H a partir do grafo bipartido G. Esta etapa leva tempo $O(n + m)$. A rede H tem $n + 2$ vértices e $n + m$ arestas.
2. Determinamos um fluxo máximo para H usando o algoritmo de Ford-Fulkerson. Como o valor do fluxo máximo é igual a $|M|$, o tamanho da maior associação, e $|M| \leq n/2$, então esta etapa leva tempo $O(n(n + m))$, que é $O(nm)$ porque G é conexo.

Portanto, temos o seguinte resultado.

Teorema 8.13: *seja G um grafo bipartido com n vértices e m arestas. Uma associação máxima em G pode ser feita em tempo* $O(nm)$.

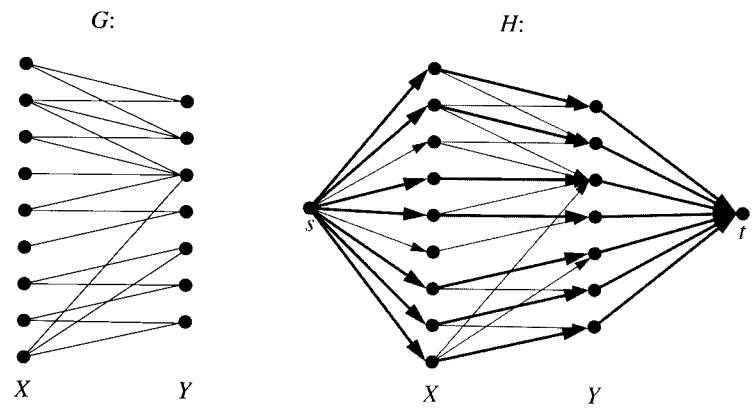


Figura 8.11 (a) Um grafo bipartido G. (b) Rede de fluxo H derivada de G e um fluxo máximo para H; as arestas mais grossas têm fluxo unitário e as outras têm fluxo zero.

8.4 Fluxo de custo mínimo

Existe uma variação do problema de fluxo máximo que se aplica em situações em que há um custo associado em mandar uma unidade de fluxo através de uma aresta. Nesta seção, estenderemos a definição de uma rede especificando um segundo peso inteiro e não-negativo $w(e)$ para cada aresta e, representando o ***custo*** da aresta e.

Dado um fluxo f, definimos o custo de f como

$$w(f) = \sum_{e \in E} w(e) f(e),$$

onde E denota o conjunto de arestas da rede. O fluxo f é dito um ***fluxo de custo mínimo*** se f tiver um custo mínimo entre todos os fluxos de valor $|f|$. ***O problema do fluxo de custo mínimo*** consiste em encontrar um fluxo máximo que tenha o menor custo entre todos os fluxos máximos. Uma variação do problema do fluxo de custo mínimo exige que encontremos um fluxo de custo mínimo com um determinado valor de fluxo. Dado um caminho aumentado π considerando um fluxo f, definimos o custo de π, denotado por $w(\pi)$, como a soma dos custos das arestas de avanço de π menos a soma dos custos das arestas de retorno de π.

8.4.1 Ciclos aumentados

Um ***ciclo aumentado*** considerando o fluxo f é um caminho aumentado cujos primeiro e último vértices são os mesmos. Em termos matemáticos, é um ciclo dirigido γ com vértices $v_0, v_1, \ldots, v_{k-1}, v_k = v_0$ tal que $\Delta_f(v_i, v_{i+1}) > 0$ para $i = 0, \ldots, k-1$ (veja a Figura 8.12). As definições de capacidade residual (dada na Seção 8.2.1) e custo (dada acima) também se aplicam a um ciclo aumentado. Além disso, como é um ciclo, podemos adicionar o fluxo de um ciclo aumentado a um fluxo existente sem mudar seu valor.

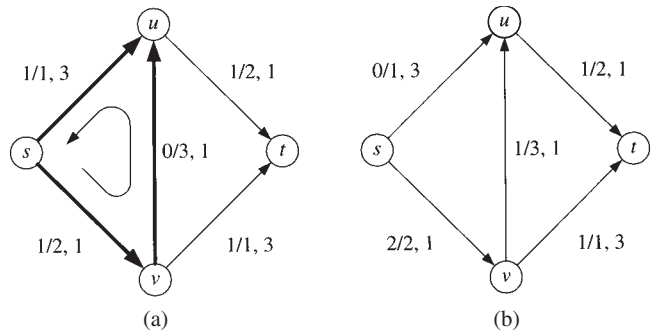


Figura 8.12 (a) Rede com fluxo f, onde cada aresta e é rotulada com $f(e)/c(e), w(e)$. Temos $|f| = 2$ e $w(f) = 8$. O ciclo aumentado $\gamma = (s, v, u, s)$ é desenhado com linhas grossas. A capacidade residual de γ é $\Delta_f(\gamma) = 1$. O custo de γ é $w(\gamma) = -1$. (b) Fluxo f' obtido de f forçando-se mais uma unidade de fluxo no ciclo γ. Temos $|f'| = |f|$ e $w(f') = w(f) + w(\gamma)\Delta_f(\gamma) = 8 + (-1).\ 1 = 7$.

Adicionando o fluxo de um ciclo aumentado

O lema a seguir é análogo ao Lema 8.5, e mostra que um fluxo máximo pode ser transformado em outro fluxo máximo usando um ciclo aumentado.

Lema 8.14: *seja γ um ciclo aumentado para o fluxo f na rede N. Existe um fluxo f' para N de valor $|f'| = |f|$ e custo*

$$w(f') = w(f) + w(\gamma)\Delta_f(\gamma).$$

Deixamos a prova do Lema 8.14 como exercício (R-8.13).

Uma condição para fluxos de custo mínimo

Note que o Lema 8.14 implica que, se um fluxo f tiver um ciclo aumentado com custo negativo, então f não tem um custo mínimo. O teorema a seguir mostra que o inverso também se verifica, fornecendo uma condição para testar se um fluxo tem custo mínimo.

Teorema 8.15: *um fluxo f tem custo mínimo entre todos os fluxos de valor $|f|$ se e somente se não houver um ciclo aumentado de custo negativo considerando f.*

Prova: a parte do "somente se" decorre do Lema 8.14. Para provar a parte do "se", suponha que o fluxo f não tem um custo mínimo e g seja um fluxo de valor $|f|$ com custo mínimo. O fluxo g pode ser obtido de f por uma série de aumentos através de ciclos aumentados. Como o custo de g é menor do que o custo de f, ao menos um desses ciclos deve ter um custo negativo. ■

Uma abordagem algorítmica para determinar fluxos de custo mínimo

O Teorema 8.15 sugere um algoritmo para o problema do fluxo de custo mínimo baseado no aumento do fluxo repetidamente através de ciclos de custo negativo. Primeiramente encontramos um fluxo máximo f^* usando o algoritmo de Ford-Fulkerson ou Edmonds-Karp. Em seguida determinamos se f^* admite um ciclo aumentado com custo negativo. O algoritmo de Bellman-Ford (Seção 7.1.2) pode ser usado para encontrar um ciclo negativo em tempo $O(mn)$. Seja w^* o custo total do fluxo máximo inicial f^*. Após cada execução do algoritmo de Bellman-Ford, o custo do fluxo diminui em pelo menos uma unidade. Portanto, iniciando com o fluxo máximo f^*, podemos determinar um fluxo máximo com mínimo custo em tempo $O(w^*mn)$. Assim, temos o seguinte:

Teorema 8.16: *dada uma rede de fluxo N com n vértices e custos associados a suas m arestas, bem como um fluxo máximo f^*, podemos determinar um fluxo máximo de custo mínimo em tempo $O(w^*mn)$, onde w^* é o custo total de f^*.*

Podemos fazer muito mais do que isto, no entanto, sendo mais cuidadosos na forma de determinar ciclos aumentados, como mostraremos no restante desta seção.

8.4.2 Caminhos mínimos sucessivos

Nesta seção, apresentaremos um método alternativo para determinar um fluxo de custo mínimo. A idéia é começar com um fluxo vazio e aumentá-lo até um fluxo máximo através de uma série de aumentos através de caminhos de custo mínimo. O teorema a seguir fornece o fundamento desta abordagem.

Teorema 8.17: *seja f um fluxo de custo mínimo e f' o fluxo obtido aumentando-se f através de um caminho aumentado π com custo mínimo. O fluxo f' é um fluxo de custo mínimo.*

Prova: a prova é ilustrada na Figura 8.13.

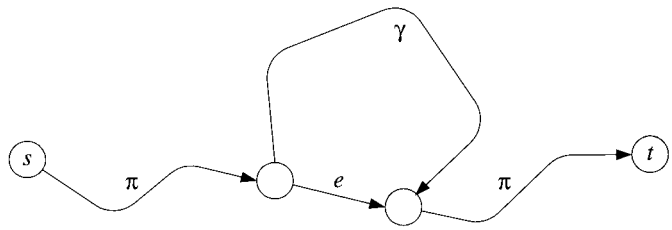


Figura 8.13 Ilustração da prova do Teorema 8.17.

Suponha que, para obtermos uma contradição, f' não tem um custo mínimo. Pelo Teorema 8.15, f' tem um ciclo aumentado γ com custo negativo. O ciclo γ deve ter ao menos uma aresta e em comum com o caminho π e percorre e na direção oposta à de π, pois em caso contrário γ seria um ciclo aumentado de custo negativo considerando o fluxo f, o que é impossível, pois f tem custo mínimo. Considere o caminho $\hat{\pi}$ obtido de π substituindo a aresta e por $\gamma - e$. O caminho $\hat{\pi}$ é um caminho aumentado considerando o fluxo f. O caminho $\hat{\pi}$ tem custo

$$w(\hat{\pi}) = w(\pi) + w(\gamma) < w(\pi) \ .$$

Isso contradiz a hipótese de que π é um caminho aumentado de custo mínimo com relação ao fluxo f. ■

Iniciando de um fluxo inicial vazio, podemos determinar um fluxo máximo de custo mínimo pela aplicação repetida do Teorema 8.17 (veja a Figura 8.14). Dado o fluxo corrente f, associamos um peso às arestas do grafo residual R_f como segue (lembre-se da definição de grafo residual da Seção 8.2.2.). Para cada aresta e dirigida de u a v na rede original, a aresta de R_f de u para v, denotada por (u, v), tem peso $w(u, v) = w(e)$, enquanto a aresta (v, u) de v para u tem peso $w(v, u) = -w(e)$. A determinação de um caminho mínimo em R_f pode ser feita usando-se o algoritmo de Bellman-Ford (da Seção 7.1.2), já que pelo Teorema 8.15 R_f não tem ciclos de custo negativo. Assim, obtemos um algoritmo de tempo pseudopolinomial (Seção 5.3.3) que determina um fluxo máximo f^* de custo mínimo em tempo $O(|f^*|\, mn)$.

Um exemplo da execução do algoritmo acima pode ser visto na Figura 8.14.

8.4.3 Pesos modificados

Podemos reduzir o tempo para determinação do caminho mínimo alterando os pesos no grafo residual R_f de forma que eles sejam todos não-negativos. Após a modificação, podemos usar o algoritmo de Dijkstra, que é executado em tempo $O(m \log(n))$, em vez do algoritmo de Bellman-Ford, que roda em tempo $O(mn)$.

Descreveremos agora a modificação dos pesos das arestas. Seja f o fluxo de custo mínimo corrente. Denotamos com $d_f(v)$ a ***distância*** do vértice v à fonte s em R_f, definida como o menor peso de um caminho de s a v em R_f (o custo de um caminho aumentado da fonte s ao vértice v). Note que esta definição de distância é diferente da usada na Seção 8.2.4 para o algoritmo de Edmonds-Karp.

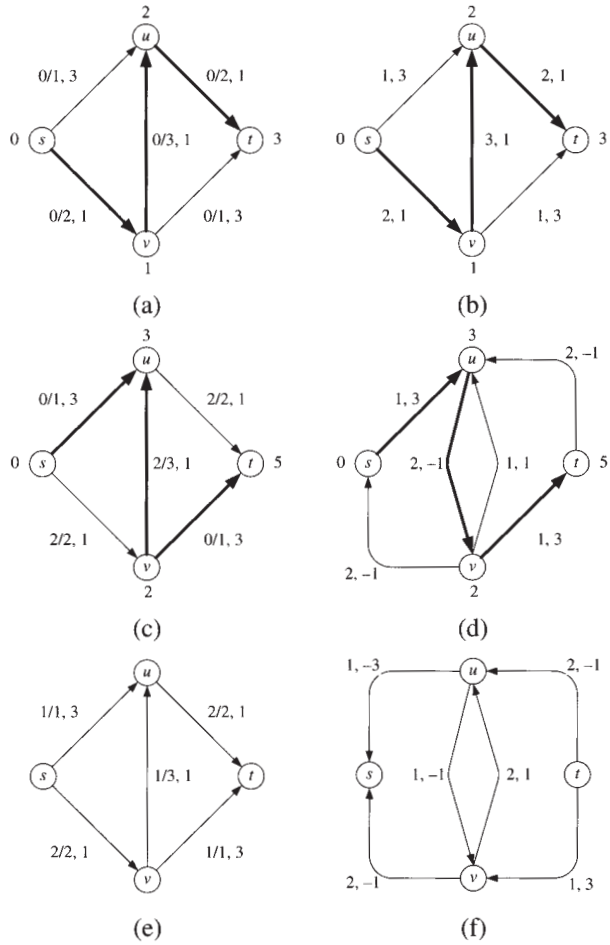


Figura 8.14 Exemplo da determinação de um fluxo de custo mínimo através de sucessivos aumentos por caminho mínimo. A cada passo, mostramos a rede à esquerda e a rede residual à direita. Os vértices são rotulados com suas distâncias da fonte. Na rede, cada aresta e é rotulada com $f(e)/c(e)$, $w(e)$. Na rede residual, cada aresta é rotulada com sua capacidade residual e custo (arestas com capacidade residual zero são omitidas). Caminhos aumentados são representados com linhas grossas, e um fluxo de custo mínimo é determinado com dois aumentos. No primeiro aumento, duas unidades de fluxo são adicionadas pelo caminho (s, v, u, t) e, no segundo aumento, uma unidade de fluxo é adicionada pelo caminho (s, u, v, t).

Seja g o fluxo obtido de v pelo aumento de f através de um caminho de custo mínimo. Definimos novos pesos w' para as arestas de R_f, como segue (veja a Figura 8.15):

$$w'(u,v) = w(u,v) + d_f(u) + d_f(v).$$

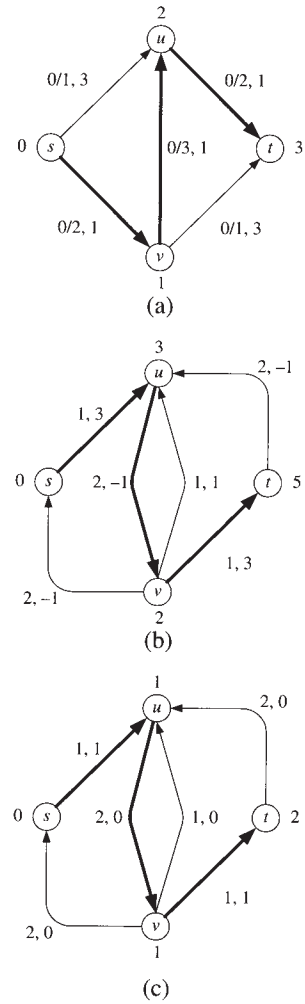


Figura 8.15 Modificação dos custos das arestas na determinação de um fluxo de custo mínimo através de sucessivos aumentos por caminhos mínimos. (a) Rede de fluxo N_f com fluxo inicial nulo f e caminho aumentado mínimo $\pi_1 = (s, v, u, t)$ com custo $w_1 = w(\pi_1) = 3$. Cada vértice é rotulado com sua distância d_f até a fonte (igual à Figura 8.14.b). (b) Rede residual R_g após o fluxo f ser aumentado em duas unidades através do caminho π e o caminho $\pi_2 = (s, u, v, t)$ com custo $w(\pi_1) = 5$ (igual à Figura 8.14.d). (c) Rede residual R_g com pesos modificados nas arestas. O caminho π_2 ainda é um caminho mínimo; no entanto, seu custo é decrescido em w_1.

Lema 8.18: *para cada aresta* (u, v) *da rede residual* R_g, *temos*

$$w'(u, v) \geq 0.$$

Um caminho mínimo em R_g com os pesos das arestas modificados é também um caminho mínimo com os pesos originais.

Prova: diferenciamos dois casos.

Caso 1: a aresta (u, v) existe em R_f.
Neste caso a distância $d_f(v)$ de v para s não é maior do que a distância $d_f(u)$ de u para s mais o peso $w(u, v)$ da aresta (u, v), ou seja,

$$d_f(v) \leq d_f(u) + w(u,v).$$

Assim, temos

$$w'(u,v) \geq 0.$$

Caso 2: a aresta (u, v) não existe em R_f.
Neste caso, (v, u) deve ser uma aresta do caminho aumentado usado para obter o fluxo g a partir do fluxo f e temos

$$d_f(u) = d_f(v) + w(v,u).$$

Como $w(v, u) = -w(u, v)$, temos

$$w'(u,v) = 0.$$

Dado um caminho π em R_g de s a t, o custo $w'(\pi)$ de π com relação aos pesos modificados das arestas difere do custo $c(\pi)$ de π por uma constante:

$$w'(\pi) = w(\pi) + d_f(s) - d_f(t) = w(\pi) - d_f(t).$$

Assim, um caminho mínimo em R_g com relação aos pesos originais é também um caminho mínimo com relação aos pesos modificados. ■

O algoritmo completo para determinar um fluxo de custo mínimo usando o método dos caminhos mínimos sucessivos é dado no Algoritmo 8.16 (MinCostFlow).

Algoritmo MinCostFlow(N):
  ***Entrada*:** rede de fluxo ponderada $N = (G, c, w, s, t)$.
  ***Saída*:** um fluxo máximo com custo mínimo f para N.
  para cada aresta $e \in N$ **faça**
    $f(e) \leftarrow 0$
  para cada vértice $v \in N$ **faça**
    $d(v) \leftarrow 0$
  stop $\leftarrow$ **falso**
  repita
    determine a rede residual ponderada R_f
    para cada aresta $(u, v) \in R_f$ **faça**
      $w'(u, v) \leftarrow w(u, v) + d(u) - d(v)$

execute o algoritmo de Dijkstra em R_f usando os pesos w'
para cada vértice $v \in N$ **faça**
  $d(v) \leftarrow$ distância de v a s em R_f
se $d(t) < +\infty$ **então**
  {π é um caminho aumentado com relação a f}
  {Determine a capacidade residual $\Delta_f(\pi)$ de π}
  $\Delta \leftarrow +\infty$
  para cada aresta $e \in \pi$ **faça**
    se $\Delta_f(e) < \Delta$ **então**
    $\Delta \leftarrow \Delta_f(e)$
  {Coloque mais $\Delta = \Delta_f(\pi)$ unidades de fluxo pelo caminho π}
  para cada aresta $e \in \pi$ **faça**
    se e for uma aresta de avanço **então**
      $f(e) \leftarrow f(e) + \Delta$
    senão
      $f(e) \leftarrow f(e) - \Delta$ {e é uma aresta de retorno}
senão
  stop $\leftarrow$ **verdadeiro** {f é um fluxo máximo de custo mínimo}
até *stop*

Algoritmo 8.16 Algoritmo para determinar um fluxo de custo mínimo usando o método dos caminhos mínimos sucessivos.

Resumimos esta seção com o teorema a seguir:

Teorema 8.19: *um fluxo máximo f de custo mínimo para uma rede com n vértices e m arestas pode ser determinada em tempo* $O(|f|m \log(n))$.

8.5 Exemplo em Java: fluxo de custo mínimo

Nesta seção, apresentaremos porções de uma implementação em Java do Algoritmo 8.16 (MinCostFlow), que determina um fluxo de custo mínimo usando aumentos sucessivos através de caminhos de custo mínimo. A classe abstrata MinCostFlowTemplate, baseada no padrão de métodos de *templates*, implementa a funcionalidade central do algoritmo. Qualquer classe concreta implementando a classe abstrata MinCost FlowTemplate deve reescrever os métodos cost(e) e capacity(e) para retornar o custo e a capacidade da aresta e específica da aplicação. As variáveis instanciadas e os métodos abstratos cost(e) e capacity(e) da classe MinCostFlowTemplate são mostrados no Trecho de Código 8.18. A computação básica de um caminho mínimo usando o método doOneIteration( ) é mostrada no Trecho de Código 8.19.

O algoritmo pode ser usado passo a passo ou de uma única vez. Os dois métodos execute mostrados no Trecho de Código 8.20 executam o algoritmo na rede de fluxo fornecida. O algoritmo será executado até que um dado valor seja atingido para o fluxo ou até que não haja mais caminhos de aumento da fonte (variável instanciada source_) até o destino (variável instanciada dest_). Opcionalmente o algoritmo pode ser

executado com um caminho aumentado de cada vez, inicializando-se o algoritmo através do método init(G, s, t) (Trecho de Código 8.21) e repetidamente chamando doOneIteration( ). Métodos auxiliares são mostrados no Trecho de Código 8.22. Para remover todos os objetos auxiliares usados pelo algoritmo, o método cleanup( )(não mostrado) pode ser chamado.

O Trecho de Código 8.23 contém métodos para trabalhar com um grafo residual. O método distance(v) retorna a distância do vértice v até a fonte do grafo residual. O método residualWeight(e) retorna o peso modificado da aresta e. O método isAtCapacity(v, e) é usado para determinar se a aresta e tem capacidade residual nula quando percorrida pelo ponto final, o vértice v. Adicionalmente, o método doOneIteration( ) retorna um iterador para as arestas do caminho aumentado corrente. Os resultados são reportados pelos métodos flow(e) e maximumFlow( ) (não mostrado).

A classe MinCostFlowTemplate usa uma classe auxiliar MinCostFlowDijkstra (não mostrada) para determinar caminhos mínimos com o algoritmo de Dijkstra. A classe MinCostFlowDijkstra é uma especialização do algoritmo genérico de caminhos mínimos de Dijkstra fornecido pela biblioteca JDSL [195] (similar à classe dada nos Trecho de Código 7.23-7.25). Os métodos principais de MinCostFlowDijkstra são os seguintes: o método weight(e) retorna o peso modificado da aresta e, como foi determinado pelo método residualWeight(e) da classe MinCostFlowTemplate, o método incidentEdges(v) também é sobrescrito para considerar somente as arestas incidentes no vértice v com capacidade residual não-nula (este método, na verdade, "remove" arestas saturadas do grafo residual). Adicionalmente, através do uso de decorações, MinCostFlowDijkstra mantém controle dos gargalos de fluxo nos caminhos mínimos correntes passando por cada vértice. Assim, ao final de cada execução de MinCostFlowDijkstra, conhecemos o caminho de menor custo da fonte ao destino, bem como a maior quantidade de fluxo que pode ser colocada por aquele caminho.

```
/**
 * Implementação do algoritmo de fluxo de custo mínimo baseado
 * em aumentos sucessivos através de caminhos de custo mínimo. O
 * algoritmo assume que o grafo não tem arestas de peso negativo.
 * A implementação usa o método de templates.
 */
public abstract class MinCostFlowTemplate {

  // variáveis instanciadas
  protected MinCostFlowDijkstra dijkstra_;
  protected InspectableGraph graph_;
  protected Vertex source_;
  protected Vertex dest_;
  protected boolean finished_;
  protected int maximumFlow_;
  protected int targetFlow_;

  // variáveis constantes
  public final int ZERO = 0;
  public final int INFINITY = Integer. MAX_VALUE;
```

```
// decorações de nodos
private final Object FLOW          = new Object( );
private final Object DISTANCE      = new Object( );

/**
 * Retorna o custo de uma aresta específica. Deve ser sobrescrito para
 * cada implementação específica.
 * @param e Aresta cujo custo é desejado
 * @return int custo da aresta e (não-negativo) */
protected abstract int cost(Edge e);

/**
 * Retorna a capacidade para uma aresta específica. Deve ser sobrescrito para
 * cada implementação específica.
 * @param e Aresta cuja capacidade é desejada
 * @return int capacidade da aresta e (não-negativa) */
protected abstract int capacity(Edge e);
```

Trecho de Código 8.18 Variáveis instanciadas e métodos abstratos para a classe MinCostFlowTemplate.

```
/**
 * Realiza uma iteração do algoritmo. O EdgeIterator
 * retornado contém as arestas que formam o caminho associado da fonte
 * ao destino.
 *
 * @return EdgeIterator das arestas consideradas no caminho aumentado */
public final EdgeIterator doOneIteration( )
 throws jdsl.graph.api. InvalidEdgeException {
 EdgeIterator returnVal;
 runDijkstraOnResidualNetwork( );
 updateDistances( );
 // verifica se um caminho aumentado existe
 if (distance(dest_) < INFINITY) {
      EdgeIterator pathIter = dijkstra_.reportPath( );
      int maxFlow = dijkstra_.reportPathFlow(dest_);
      maximumFlow_ += maxFlow;
      // coloca MaxFlow através do caminho
      while (pathIter.hasNext( )) {
        // verifica se é uma aresta de avanço
        Edge e = pathIter.nextEdge( );

        if (isForwardEdge(e)) {
           setFlow(e, flow(e) + maxFlow);
        } else {
          setFlow(e, flow(e) – maxFlow);
        }
      }
      pathIter. reset( );
```

```
        returnVal = pathIter;
    }   else {
        finished();
        returnVal = new EdgeIteratorAdapter(new ArrayObjectIterator(new Object[0]));
    }
    return returnVal;
}
```

Trecho de Código 8.19 O método para determinar um caminho aumentado de custo mínimo na classe MinCostFlowTemplate.

```
  /**
 * Método auxiliar para executar continuamente iterações do algoritmo
 * até que ele termine.*/
protected final void runUntil( ) {
    while (shouldContinue( )) {
        doOneIteration( );
    }
}
/**
 * Executa o algoritmo que determina o fluxo máximo entre a fonte e o
 * destino no Grafo g.
 *
 * @param g um Grafo
 * @param source fonte do fluxo
 * @param dest destino do fluxo*/
public final void execute(InspectableGraph g, Vertex source, Vertex dest)
    throws InvalidVertexException {
    init(g, source, dest);
    runUntil( );
}
/**
 * Executa o algoritmo, que será executado até que o fluxo desejado seja alcançado
   ou * até que nenhum fluxo seja possível.
 *
 * @param um Grafo
 * @param source fonte do fluxo
 * @param dest destino do fluxo */
public final void execute(InspectableGraph g, Vertex source, Vertex dest, int target)
        throws InvalidVertexException {
        targetFlow_ = target;
        execute(g, source, dest);
}
```

Trecho de Código 8.20 Métodos para controlar a execução do algoritmo de fluxo de custo mínimo na classe MinCostFlowTemplate.

```
/**
```

```
/**
 * Inicializa o algoritmo. Inicializa as variáveis locais e
 * os valores default para as decorações. O algoritmo de Dijkstra
 * também é inicializado.
 *
 * @param g um Grafo
 * @param source fonte do fluxo
 * @param dest destino do fluxo      */
public void init(InspectableGraph g, Vertex source, Vertex dest)
      throws InvalidVertexException {
      if( !g.contains( source ) )
        throw new InvalidVertexException( source + " not contained in" + g );
      if( !g.contains( dest ) )
        throw new InvalidVertexException( dest + "not contained in" + g);
      graph_ = g;
      source_ = source;
      dest_ = dest;
      finished_ = false;
      maximumFlow_ = ZERO;
      targetFlow_ = INFINITY;
      // inicializa Dijkstra
      dijkstra_ = new MinCostFlowDijkstra( );
      dijkstra_.init(g, source);
      // inicializa valores default
      VertexIterator vertexIter = vertices( );
      while (vertexIter.hasNext( )) {
        Vertex u = vertexIter.nextVertex( );
        setDistance(u, ZERO);
      }
      EdgeIterator edgeIter = edges( );
      while (edgeIter. hasNext( )) {
        Edge e = edgeIter.nextEdge( );
        setFlow(e, ZERO);
      }
  }
```

Trecho de Código 8.21 Método de inicialização da classe MinCostFlowTemplate.

```
/**
 * Método auxiliar para copiar todas as distâncias de vértices
 * de uma execução do algoritmo de Dijkstra para decorações locais
 * de forma que possam ser usadas para determinar o grafo residual para a
 * próxima execução do algoritmo de Dijkstra.*/
protected void updateDistances( ) {
  // copia distâncias da rede residual para nossa rede
  VertexIterator vertexIter = vertices( );
  while (vertexIter.hasNext( )) {
        Vertex v = vertexIter.nextVertex( );
```

```
        try {
            setDistance(v, dijkstra_.distance(v));
        } catch (InvalidQuerylException iqe) {
            // vértice não é atingível, distância é INFINITY
            setDistance(v, INFINITY);
        }
    }
}

/**
 * Método auxiliar para executar Dijkstra na rede residual. Temos
 * certeza de limpar todas as execuções passadas ao chamar primeiro o
 * método cleanup( ).*/
protected void runDijkstraOnResidualNetwork( ) {
  dijkstra_.cleanup( );
  dijkstra_.execute (graph_, source_, dest_);
}
/**
 * Método auxiliar chamado uma vez quando o algoritmo termina de executar.     */
protected void finished( ) {
   finished_ = true;
}
```

Trecho de Código 8.22 Métodos auxiliares da classe MinCostFlowTemplate.

```
/**
 * Retorna a distância de um vértice à fonte
 * @param v um vértice
 * @return a distância do vértice v até a fonte
 * @throws InvalidQueryException se v ainda não foi atingido    */
public final int distance(Vertex v) throws InvalidQueryException {
   try {
        return (Integer)v.get(DISTANCE)).intValue( );
   }
   catch (InvalidAttributeException iae) {
        throw new InvalidQueryException(v+" has not been reached yet");
   }
}
/**
 * Retorna o peso modificado da aresta e no grafo residual corrente.
 * Pode ser calculado rapidamente porque a informação de distância
 * somente é atualizada após cada iteração do algoritmo.
 * @param e Aresta para a qual se deve encontrar o peso modificado
 * @return int peso residual de e */
public final int residualWeight(Edge e) {
   // usa o valor absoluto porque se percorremos a aresta de trás para a
   // frente temos w(u,v) = – w(v,u)
   return Math.abs( cost(e) +
                                distance(graph_.origin(e)) –
                                distance(graph_.destination (e)) );
```

```
}
/**
 * Determina se a aresta e tem capacidade residual nula quando
 * percorrida começando no vértice v.
 * @param v um Vértice a partir do qual a aresta está sendo considerada.
 * @param e Aresta a verificar
 * @return booleano true se a aresta não tem capacidade residual, false se tiver.*
public final boolean isAtCapacity(Vertex v, Edge e) {
   // arestas de avanço estão cheias se capacidade == fluxo
   if( v == graph_.origin( e ) )
        return (capacity(e) == flow(e));
   // arestas de retorno estão cheias quando fluxo == 0
   else
        return (flow(e) == 0);
}
```

Trecho de Código 8.23 Os métodos distance, residualWeight e isAtCapacity da classe MinCostFlowTemplate.

8.6 Exercícios

Reforço

R-8.1 Demonstre que, para um fluxo f, o fluxo total saindo da fonte é igual ao fluxo total chegando no destino, ou seja,

$$\sum_{e \in E^+(s)} f(e) = \sum_{e \in E^-(t)} f(e).$$

R-8.2 Responda às seguintes questões sobre o a rede de fluxo N e o fluxo f mostrados na Figura 8.6a:

- Quais são as arestas de avanço do caminho aumentado π? Quais são as arestas de retorno?
- Quantos caminhos aumentados existem considerando o fluxo f? Para cada caminho, liste a seqüência de vértices do caminho e a capacidade residual do caminho.
- Qual o valor do fluxo máximo em N?

R-8.3 Construa um corte mínimo para a rede mostrada na Figura 8.4 usando o método da prova do Lema 8.6.

R-8.4 Ilustre a execução do algoritmo de Ford-Fulkerson na rede de fluxo da Figura 8.2.

R-8.5 Desenhe uma rede de fluxo com nove vértices e 12 arestas. Ilustre a execução do algoritmo de Ford-Fulkerson sobre ela.

R-8.6 Ache um corte mínimo na rede de fluxo da Figura 8.8a.

R-8.7 Demonstre que, dado um fluxo máximo em uma rede de fluxo de m vértices, um corte mínimo de N pode ser determinado em tempo $O(m)$.

R-8.8 Encontre duas associações máximas para o grafo bipartido da Figura 8.11a que sejam diferentes da associação máxima da Figura 8.11b.

R-8.9 Seja G um grafo completo bipartido tal que $|X| = |Y| = n$ e para cada par de vértices $x \in X$ e $y \in Y$, exista uma aresta unindo x e y. Demonstre que G tem $n!$ diferentes associações máximas.

R-8.10 Ilustre a execução do algoritmo de Ford-Fulkerson na rede de fluxo da Figura 8.11b.

R-8.11 Ilustre a execução do algoritmo de Edmonds-Karp na rede de fluxo da Figura 8.8a.

R-8.12 Ilustre a execução do algoritmo de Edmonds-Karp na rede de fluxo da Figura 8.2.

R-8.13 Forneça um prova do Lema 8.14.

R-8.14 Ilustre a execução do algoritmo de fluxo de custo mínimo baseado em aumentos sucessivos através de ciclos de custo negativo para a rede de fluxo da Figura 8.14a.

R-8.15 Ilustre a execução do algoritmo de fluxo de custo mínimo baseado em aumentos sucessivos através de caminhos de custo mínimo para a rede de fluxo da Figura 8.2, onde o custo de uma aresta (u, v) é dado por $|deg(u) - deg(v)|$.

R-8.16 O Algoritmo 8.16 (MinCostFlow) é um algoritmo de tempo pseudopolinomial?

Criatividade

C-8.1 Qual o tempo de execução no pior caso do algoritmo de Ford-Fulkerson se todas as capacidades das arestas são limitadas por uma constante?

C-8.2 Melhore o limite do Lema 8.9 mostrando que existem no máximo $mn/4$ aumentos no algoritmo de Edmonds-Karp.

Dica: use $d_f(u, t)$ além de $d_f(s, v)$.

C-8.3 Seja N uma rede de fluxo com n vértices e m arestas. Demonstre como computar um caminho aumentado com a maior capacidade residual em tempo $O((n + m)\log(n))$.

C-8.4 Demonstre que o algoritmo de Ford-Fulkerson é executado em tempo $O(m^2\log(n)\log(|f^*|))$ se a cada iteração o caminho aumentado com a maior capacidade residual for escolhido.

C-8.5 Você deseja aumentar o fluxo máximo de uma rede de fluxo o máximo possível, mas só pode aumentar a capacidade de uma aresta.

a. Como você encontra essa aresta? (Forneça o pseudocódigo.) Você pode assumir a existência de algoritmos para determinar o fluxo máximo e o corte mínimo. Qual o tempo de execução de seu algoritmo?

b. É sempre possível encontrar uma aresta como essa? Justifique sua resposta.

C-8.6 Dada uma rede de fluxo N e um fluxo máximo f em N, suponha que a capacidade de uma aresta e em N seja diminuída em um, e seja N' a rede resultante. Forneça um algoritmo para determinar o fluxo máximo na rede N' modificando f.

C-8.7 Forneça um algoritmo que determina, em tempo $O(n + m)$, se um grafo com n vértices e m arestas é bipartido.

C-8.8 Forneça um algoritmo que determina qual o maior número de caminhos sem arestas em comum entre dois vértices s e t de um grafo não-dirigido.

C-8.9 Uma companhia de táxis recebe chamadas de n lugares. Existem m táxis disponíveis, com $m \geq n$, e a distância do táxi i ao lugar j é d_{ij}. Forneça um algoritmo para determinar o envio de n táxis para os n lugares minimizando a distância total.

C-8.10 Forneça um algoritmo para determinar um fluxo de valor máximo sujeito às seguintes restrições adicionais:

a. Cada aresta e tem um limite inferior $l(e)$ no fluxo passando por ela.

b. Existem múltiplas fontes e destinos, e o valor do fluxo é calculado como o fluxo total saindo de todas as fontes (igual ao fluxo total entrando em todos os destinos).

C-8.11 Demonstre que, em uma rede de fluxo com capacidades não-inteiras, o algoritmo de Ford-Fulkerson pode não terminar.

Projetos

P-8.1 Projete e implemente uma *applet* que anima o algoritmo de Ford-Fulkerson. Seja criativo na forma de ilustrar o aumento de fluxo, a capacidade residual e o próprio fluxo.

P-8.2 Implemente o algoritmo de fluxo de Ford-Fulkerson usando três métodos diferentes para encontrar caminhos aumentados. Realize uma comparação experimental entre os métodos.

P-8.3 Implemente o algoritmo para determinar a associação máxima bipartida. Mostre como reutilizar um algoritmo para determinar o fluxo máximo.

P-8.4 Implemente o algoritmo de Edmonds-Karp.

P-8.5 Implemente o algoritmo de fluxo de custo mínimo baseado em aumentos sucessivos através de ciclos de custo negativo.

P-8.6 Implemente o algoritmo de fluxo de custo mínimo baseado em aumentos sucessivos através de caminhos de custo mínimo. Implemente a variação que usa o algoritmo de Bellman-Ford e a variação que modifica os custos e usa o algoritmo de Dijkstra.

Notas

O algoritmo de fluxo em rede de Ford-Fulkerson (8.2.2) é descrito em seu livro [71]. Edmonds e Karp [65] descrevem dois métodos para determinar caminhos aumentados que fazem com que o algoritmo de Ford-Fulkerson seja mais rápido: caminho aumentado mínimo (Seção 8.2.4) e caminho aumentado com máxima capacidade residual (Exercício C-8.4). O algoritmo de fluxo de custo mínimo baseado em aumentos sucessivos através de caminhos de custo mínimo (Seção 8.4.2) também foi desenvolvido por Edmonds e Karp [65].

O leitor interessado em mais detalhes sobre algoritmos para grafos e fluxo em redes pode consultar os livros de Ahuja, Magnanti e Orlin [9], Cormen, Leiserson e Rivest [55], Even [68], Gibbons [77], Mehlhorn [149] e Tarjan [200], e o capítulo escrito por van Leeuwen [205].

Dan Polivy desenvolveu a implementação do algoritmo de fluxo com custo mínimo tratado na Seção 8.5.

Parte

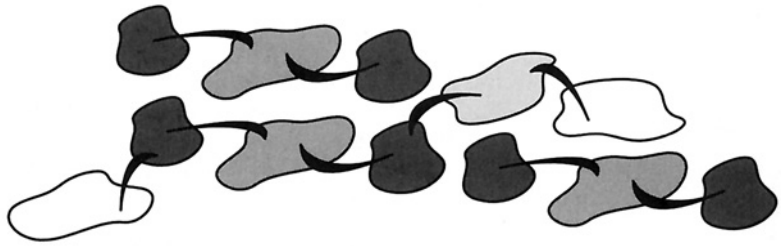
Algoritmos para Internet

III

Capítulo

Processamento de texto

9



9.1 Cadeias de caracteres e pesquisa de padrões 417

9.1.1 Operações sobre cadeias de caracteres 417

9.1.2 Procura de padrões por força bruta 418

9.1.3 O algoritmo Boyer-Moore 420

9.1.4 O algoritmo Knuth-Morris-Pratt 423

9.2 *Tries* 426

9.2.1 *Tries* padrão 426

9.2.2 *Tries* comprimidos 429

9.2.3 *Tries* de sufixos 431

9.2.4 Mecanismos de busca 434

9.3 Compressão de textos. 435

9.3.1 O algoritmo de codificação de Huffman. 437

9.3.2 O método guloso revisitado 437

9.4 Teste de similaridade de textos 438

9.4.1 O problema da maior subseqüência comum 438

9.4.2 Aplicação da programação dinâmica no problema da LCS 439

9.5 Exercícios 442

O processamento de documentos está se tornando uma das funções dominantes dos computadores. Computadores são usados para editar documentos, pesquisar documentos, transportar documentos pela Internet e apresentar documentos em telas e impressoras. Pesquisas na Web estão se tornando aplicações importantes e significativas para os computadores, e muitas das computações essenciais em todo esse processamento de documentos envolvem cadeias de caracteres e procura de padrões. Por exemplo, os formatos de documentos para Internet HTML e XML são primordialmente formatos de textos, com rótulos especiais adicionados para comportar conteúdo multimídia. Dar significado aos muitos *terabytes* de informação na Internet requer um trabalho considerável de processamento de texto.

Neste capítulo, estudamos vários algoritmos fundamentais para processamento de texto que realizam importantes operações sobre cadeias de caracteres rapidamente. Damos atenção particular aos algoritmos de pesquisa em cadeias de caracteres e procura de padrões, pois esses freqüentemente podem ser gargalos no desempenho de muitas aplicações que processam documentos. Também estudaremos alguns tópicos algorítmicos envolvidos em processamento de texto.

Algoritmos para processamento de texto operam primariamente sobre cadeias de caracteres. A terminologia e a notação para essas cadeias, como usadas neste capítulo, são bastante intuitivas, e representar uma cadeia como um arranjo de caracteres é bastante simples e eficiente. Portanto, não gastaremos tempo com a representação de cadeias de caracteres. Mesmo assim, o processamento de texto freqüentemente envolve um método interessante para procura de padrões, cujos algoritmos estudaremos na Seção 9.1.

Na Seção 9.2, estudaremos a estrutura de dados chamada *trie*, que é uma estrutura baseada em árvore que permite uma procura rápida em uma coleção de cadeias de caracteres.

Estudaremos um importante problema de processamento de texto na Seção 9.3, o problema de comprimir um documento de forma que ele ocupe menos espaço para armazenamento ou possa ser transmitido mais eficientemente em uma rede.

O último problema de processamento de texto que estudaremos, na Seção 9.4, trata de determinar a similaridade entre dois documentos. Todos esses problemas são tópicos que freqüentemente surgem em aplicações na Internet, como mecanismos de procura, distribuição de documentos e recuperação de informação.

Além de ter aplicações interessantes, os tópicos deste capítulo também salientam alguns importantes padrões de projeto de algoritmos (veja o Capítulo 5). Em particular, na seção sobre procura de padrões, discutiremos o ***método da força bruta***, que é muitas vezes ineficiente mas tem larga aplicação. Para compressão de textos, estudaremos a aplicação do ***método guloso*** (Seção 5.1), que freqüentemente permite que obtenhamos soluções aproximadas para problemas difíceis e que, para alguns problemas (como na compressão de textos), fornecem algoritmos ótimos. Finalmente, na discussão sobre similaridade de textos, fornecemos outra aplicação de ***programação dinâmica*** (Seção 5.3), que pode ser aplicada em instâncias especiais para resolver em tempo polinomial problemas que a princípio parecem requerer tempo exponencial para resolução.

9.1 Cadeias de caracteres e pesquisa de padrões

Documentos textuais são onipresentes na moderna computação, pois são usados para comunicar e publicar informação. Na perspectiva do projeto de algoritmos, esses documentos podem ser vistos como simples cadeias de caracteres. Ou seja, eles podem ser abstraídos em uma seqüência de caracteres que definem o seu conteúdo. Portanto, realizar procuras e operações de processamento neste tipo de dado requer que tenhamos métodos eficientes para trabalhar com cadeias de caracteres.

9.1.1 Operações sobre cadeias de caracteres

No núcleo dos algoritmos para processamento de texto estão métodos para tratar com cadeias de caracteres. Cadeias de caracteres podem surgir de uma variedade de origens, incluindo aplicações científicas, lingüísticas e da Internet. Abaixo, alguns exemplos dessas cadeias de caracteres:

P = `"CGTAAACTGCTTTAATCAAACGC"`
R = `"U.S. Men Win Soccer World Cup!"`
S = `"http://www.wiley.com/college/goodrich/"`

A primeira cadeia de caracteres, P, tem origem em aplicações de pesquisa de DNA, enquanto a última, S, é o endereço (URL) do *site* da Web que acompanha este livro. A cadeia de caracteres R é uma manchete fictícia. Nesta seção, apresentamos algumas das operações úteis que são suportadas pelo TAD cadeia de caracteres para processar cadeias como essas.

Várias das operações típicas do processamento de cadeias de caracteres envolvem quebrar as cadeias longas em cadeias menores. A respeito dos pedaços que resultam deste tipo de operações, usamos o termo ***substring*** de uma cadeia de caracteres P de comprimento m para nos referir a uma cadeia da forma $P[i]P[i+1]\cdots P[j]$ para $0 \le i \le j \le m$, ou seja, a cadeia formada pelos caracteres em P do índice i ao índice j inclusive. Tecnicamente, isso significa que uma cadeia de caracteres é uma *substring* de si mesma (com $i = 0$ e $j = m - 1$); por isso, se desejarmos excluir essa possibilidade, devemos restringir a definição às substrings ***próprias***, que requerem $i > 0$ ou $j < m - 1$. Com o objetivo de simplificar a notação de referência às *substrings*, usaremos $P[i..j]$ para denotar a *substring* de P do índice i ao índice j inclusive. Ou seja,

$$P[i..j] = P[i]P[i+1]\cdots P[j]$$

Usamos a convenção de que, se $i > j$, então $P[i..j]$ é uma ***cadeia vazia***, que tem comprimento 0. Além disso, para distinguir alguns casos especiais de *substrings*, chamamos qualquer cadeia da forma $P[0..i]$ para $0 \le i \le m - 1$ de ***prefixo*** de P, e qualquer cadeia da forma $P[i..m-1]$ para $0 \le i \le m - 1$ de ***sufixo*** de P. Por exemplo, se P for a cadeia de DNA dada acima, então `"CGTAA"` é um prefixo de P, `"CGC"` é um sufixo de P e `"TTAATC"` é uma *substring* (própria) de P. Note que uma cadeia vazia é prefixo e sufixo de qualquer outra cadeia.

O problema da procura de padrões

No problema clássico da ***procura de padrões*** em cadeias de caracteres, recebemos uma cadeia de caracteres ou ***texto*** T de comprimento n e uma segunda cadeia ou ***pa-***

drão P de comprimento m e desejamos saber se P é uma *substring* de T. Ou seja, desejamos saber se existe uma *substring* de T iniciando na posição i que confere com P caracter por caracter, de forma que

$$T[i] = P[0], T[i + 1] = P[1], ..., T[i + m - 1] = P[m - 1].$$

Ou seja,

$$P = T[i...i + m - 1].$$

Assim, a saída de um algoritmo de procura de padrões poderia ser tanto uma indicação de que o padrão P não existe em T como um inteiro indicando a posição ou índice em T onde uma cadeia igual a P inicia.

Considerando uma noção bastante generalizada de uma cadeia de caracteres, tipicamente não restringimos os caracteres de T e P a um conjunto de caracteres padrão, como os conjuntos de caracteres ASCII ou Unicode. Em vez disso, usamos o símbolo genérico Σ para denotar o conjunto de caracteres ou ***alfabeto*** do qual são retirados os caracteres de T e P. Esse alfabeto Σ pode, evidentemente, ser um subconjunto dos conjuntos de caracteres ASCII ou Unicode, mas também pode ser mais genérico e até mesmo infinito, como o conjunto dos inteiros positivos. Mesmo assim, uma vez que a maior parte dos algoritmos para processamento de texto são usados em aplicações em que o conjunto de caracteres é finito, geralmente assumimos que o tamanho do alfabeto Σ, denotado por $|\Sigma|$, é uma constante finita.

Exemplo 9.1: *suponha que recebemos a cadeia abaixo*

$T =$ `"abacaabaccabacabaabb"`

e a cadeia para pesquisa

$P =$ `"abacab".`

Então, P é uma *substring* de T. Mais exatamente, $P = T[10..15]$.

Nesta seção, apresentamos três algoritmos para procura de padrões.

9.1.2 Procura de padrões por força bruta

O padrão de projeto baseado em ***força bruta*** é uma técnica poderosa para o projeto de algoritmos quando temos algo a procurar ou quando desejamos otimizar alguma função. Aplicando essa técnica em uma situação genérica, enumeramos todas as configurações possíveis das entradas envolvidas e escolhemos a melhor das configurações enumeradas.

Procura de padrões por força bruta

Aplicando esta técnica para o algoritmo de ***procura de padrões por força bruta***, derivamos o que é provavelmente o primeiro algoritmo em que se pode pensar para resolver o problema da procura de padrões – simplesmente testamos todas as possíveis colocações de P em relação a T. Esse algoritmo, mostrado no Algoritmo 9.1, é bastante simples.

Algoritmo BruteForceMatch(T, P):
  Entrada: as cadeias T (texto) com n caracteres e P (padrão) com m caracteres.
  Saída: o índice da primeira *substring* de T igual a P, ou uma indicação
    de que P não é uma *substring* de T.
  para $i \leftarrow 0$ até $n - m$ {para cada índice candidato em T} **faça**
    $j \leftarrow 0$
    enquanto ($j < m$ **e** T $[i + j] = P\ [j]$) **faça**
      $j \leftarrow j + 1$
    se $j = m$ **então**
      retorne i
  retorne "Não existe *substring* em T igual a P."

Algoritmo 9.1 Procura de padrões por força bruta.

O algoritmo de procura de padrões por força bruta não poderia ser mais simples. Ele consiste em dois laços aninhados, com o laço externo pesquisando todos os possíveis índices do padrão dentro do texto e o laço interno pesquisando cada caracter do padrão, comparando-o a seu potencial correspondente no texto. Assim, a correção do algoritmo segue imediatamente desta abordagem de procura exaustiva.

O tempo de execução da procura de padrões por força bruta no pior caso não é bom, entretanto, pois para cada índice candidato em T temos de realizar até m comparações de caracteres para descobrir que P não é igual à *substring* de T começando no índice corrente. No Trecho de Código 9.1, podemos ver que o laço externo é executado no máximo $n - m + 1$ vezes, e o laço interno é executado no máximo m vezes. Assim, o tempo de execução do método de força bruta é $O((n - m + 1)m)$, que é simplificado para $O(nm)$. Note que, quando $m = n/2$, esse algoritmo tem tempo de execução quadrático $O(n^2)$.

Na Figura 9.2, ilustramos a execução do algoritmo de força bruta para procura de padrões sobre as cadeias T e P do Exemplo 9.1.

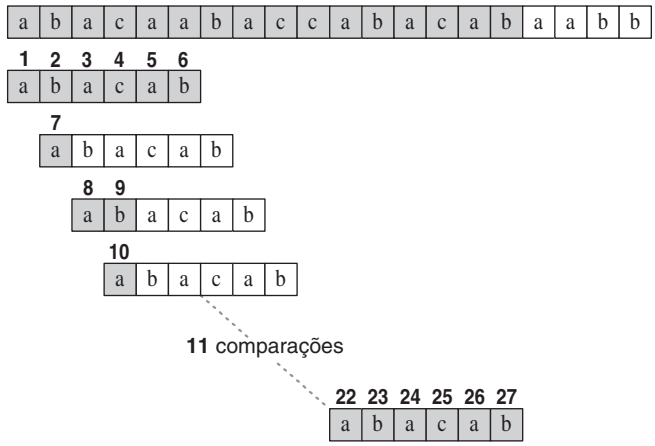


Figura 9.2 Exemplo do algoritmo de procura de padrões por força bruta. O algoritmo realiza 27 comparações de caracteres, indicadas acima pela numeração.

9.1.3 O algoritmo Boyer-Moore

Primeiramente, pensamos que é sempre necessário examinar cada caracter de T para localizar um padrão P como uma *substring*. Isso nem sempre acontece, pois o algoritmo de procura de padrões ***Boyer-Moore*** (***BM***), que estudaremos nesta seção, consegue evitar comparações entre P e uma boa parte dos caracteres em T. O único problema é que, enquanto o algoritmo de força bruta pode trabalhar com um alfabeto ilimitado, o algoritmo Boyer-Moore supõe que o alfabeto tenha tamanho finito. Ele tem melhor desempenho quando o alfabeto é de tamanho moderado e o padrão é relativamente longo.

Nesta seção, descreveremos uma versão simplificada do algoritmo original de Boyer e Moore. A idéia principal é melhorar o tempo de execução do algoritmo de força bruta adicionando a ele duas heurísticas que potencialmente podem economizar tempo:

Heurística do espelho: quando testamos uma possível colocação de P em T, começamos as comparações de forma invertida, ou seja, pelo final de P, e recuamos até o início de P.

Heurística do salto de caracteres: durante o teste de uma possível colocação de P em T, uma diferença entre o caracter $T[i] = c$ e o caracter correspondente $P[j]$ é tratada como segue: se c não estiver contido em lugar algum de P, então movemos P completamente para depois de $T[i]$ (pois $T[i]$ não pode estar em P). Em caso contrário, movemos P para a frente até que uma ocorrência do caracter c em P esteja alinhada com $T[i]$.

Formalizaremos essas heurísticas em breve, mas, de forma intuitiva, podemos perceber que elas trabalham de forma integrada. A heurística do espelho permite que a segunda heurística evite comparações entre P e grupos inteiros de caracteres em T. Neste caso, ao menos, podemos chegar ao resultado mais depressa indo de trás para a frente, pois, se encontramos uma diferença entre caracteres quando procuramos P em uma certa posição de T, então provavelmente evitaremos uma série de comparações desnecessárias se movermos P de forma significativa em relação a T através da heurística do salto de caracteres. A heurística do salto de caracteres traz grande vantagem se for aplicada cedo no teste de ocorrência de P em T.

Vamos agora definir como a heurística do salto de caracteres pode ser integrada em um algoritmo de procura de padrões em cadeias de caracteres. Para implementar essa heurística, definimos uma função $\mathsf{last}(c)$ que receba um caracter c do alfabeto e caracterize o quanto podemos avançar o padrão P se um caracter igual a c for encontrado no texto e não fizer parte do padrão. Em particular, definimos $\mathsf{last}(c)$ como

- Se c estiver em P, $\mathsf{last}(c)$ é o índice da última (mais à direita) ocorrência de c em P. Senão, convencionamos que $\mathsf{last}(c) = -1$.

Se caracteres podem ser usados como índices em arranjos, então a função last pode ser implementada facilmente como uma tabela. Deixamos o método de construção dessa tabela em tempo $O(m + |\Sigma|)$, dado P, como um exercício (R-9.6). A função last fornece toda a informação de que precisamos para realizar a heurística do salto de caracteres. No Algoritmo 9.3, mostramos o algoritmo Boyer-Moore para procura de padrões. O salto de caracteres é mostrado na Figura 9.4.

Algoritmo BMMatch(T, P):
  Entrada: as cadeias T (texto) com n caracteres e P (padrão) com m caracteres.
  Saída: o índice da primeira *substring* de T igual a P, ou uma indicação
    de que P não é uma *substring* de T.
  computar a função last
  $i \leftarrow m - 1$
  $j \leftarrow m - 1$
  repita
    se $P[j] = T[i]$ **então**
      se $j = 0$ **então**
        retorne i {achamos!}
      senão
        $i \leftarrow i - 1$
        $j \leftarrow j - 1$
    senão
        $i \leftarrow i + m - \min(j, 1 + \text{last}(T[i]))$ {salto}
        $j \leftarrow m - 1$
até $i > n - 1$
retorne "Não existe *substring* em T igual a P."

Algoritmo 9.3 Algoritmo de procura de padrões Boyer-Moore.

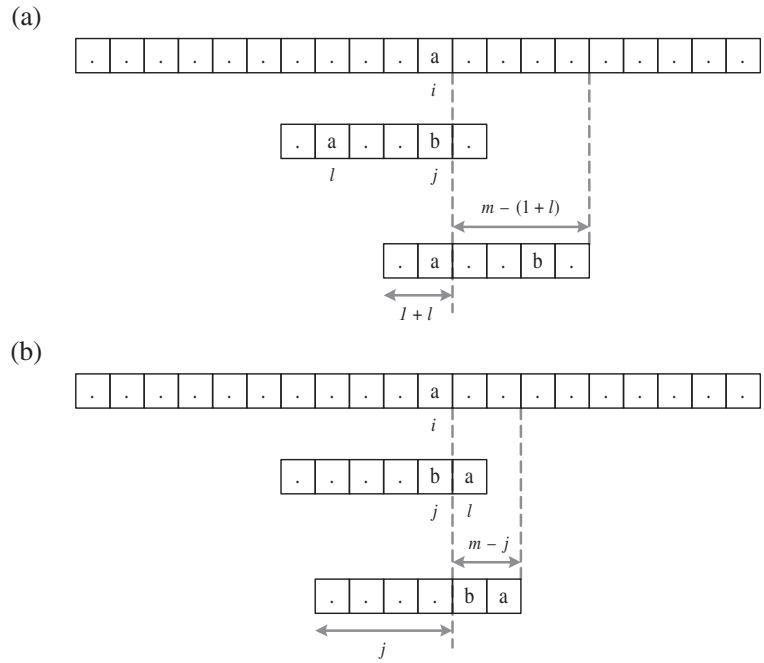
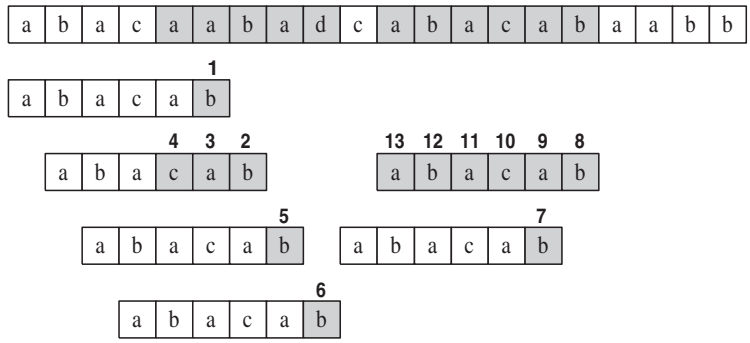


Figura 9.4 Ilustração da etapa de salto no algoritmo BM, onde l denota last($T[i]$). Diferenciamos dois casos: (a) $1 + l \leq j$, onde movemos o padrão $j - l$ unidades; (b) $j < 1 + l$, onde movemos o padrão uma unidade.

Na Figura 9.5, ilustramos a execução do algoritmo Boyer-Moore em uma entrada similar à do Exemplo 9.1.



A função last(c):

c	a	b	c	d
last(c)	4	5	3	–1

Figura 9.5 Ilustração do algoritmo Boyer-Moore de procura de padrões. O algoritmo realiza 13 comparações de caracteres, que estão numeradas.

A correção do algoritmo BM para procura de padrões vem do fato de que, cada vez que o método move o padrão, é garantido que ele não "pula" sobre um possível acerto. Isso acontece porque last(c) é o local da ***última*** ocorrência de c em P.

O tempo de execução de pior caso do algoritmo BM é $O(m + |\Sigma|)$. A computação da função last leva tempo $O(m + |\Sigma|)$, e a procura pelo padrão leva tempo $O(nm)$ no pior caso, o mesmo que o algoritmo de força bruta. Um exemplo de par texto/padrão que atinge o pior tempo é

$$T = \overbrace{aaaaaa\cdots a}^{n}$$

$$P = b\overbrace{aa\cdots a}^{m-1}.$$

O pior desempenho, no entanto, é difícil de ser obtido com texto em língua inglesa, por exemplo.

De fato, o algoritmo Boyer-Moore é freqüentemente capaz de saltar grandes partes do texto (veja a Figura 9.6.). Existe evidência experimental de que em textos de língua inglesa o número médio de comparações feitas por caracter de texto é de 0,24 para um padrão de cinco caracteres. O ganho não é tão grande em cadeias de caracteres binárias ou em padrões muito curtos e, neste caso, o algoritmo KMP discutido na seção 9.1.4 ou (para padrões muito curtos) o algoritmo de força bruta podem ser melhores.

Na realidade apresentamos uma versão simplificada do algoritmo Boyer-Moore (BM). O algoritmo BM original tem um desempenho $O(n + m + |\Sigma|)$ usando uma heurística alternativa de saltos à frente até o texto já parcialmente encontrado, sempre que ele avança o padrão mais do que a heurística de salto de caracteres apresentada aqui. Essa heurística alternativa para o salto à frente é baseada na aplicação da idéia principal do algoritmo Knuth-Morris-Pratt de procura de padrões, que discutiremos a seguir.

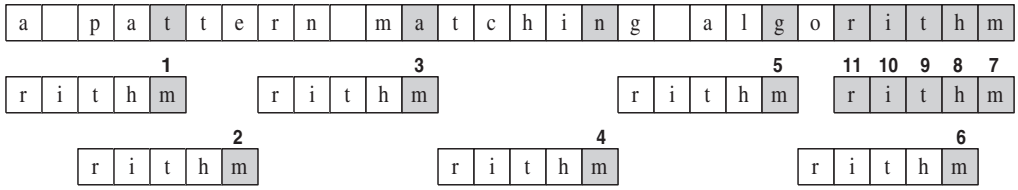


Figura 9.6 Execução do algoritmo Boyer-Moore em um par texto/padrão em inglês, onde uma aceleração significativa é obtida. Note que nem todos os caracteres do texto são examinados.

9.1.4 O algoritmo Knuth-Morris-Pratt

Estudando o desempenho de pior caso dos algoritmos de força bruta e Boyer-Moore em instâncias específicas do problema, como a mostrada no Exemplo 9.3, notamos uma fonte de ineficiência. Especificamente, podemos realizar muitas comparações enquanto testamos um posicionamento do padrão sobre o texto, mas se descobrimos um caracter do padrão que falha nesta comparação, então descartamos toda a informação adquirida pelas comparações anteriores e começamos novamente (do zero) no ponto onde o padrão for posicionado, mais à frente. O algoritmo Knuth-Morris-Pratt (KMP), discutido nesta seção, evita esse desperdício de informação e com isso atinge um tempo de execução de $O(n + m)$, que é ótimo no pior caso. Ou seja, no pior caso, qualquer algoritmo de procura de padrões terá de examinar todos os caracteres do texto e do padrão ao menos uma vez.

A função de falha

A idéia principal do algoritmo KMP é pré-processar P para calcular uma ***função de falha*** f que indica o quanto devemos avançar P de forma que possamos reutilizar as comparações realizadas anteriormente tanto quanto possível. Especificamente, a função de falha $f(j)$ é definida como o comprimento do prefixo mais longo de P, que é um sufixo de $P[1..j]$ (note que ***não*** escrevemos $P[0..j]$). Também usamos a convenção $f(0) = 0$. Mais tarde discutiremos como computar a função de falha eficientemente. A importância dessa função é que ela "codifica" *substrings* repetidas dentro do próprio padrão.

Exemplo 9.2: *considere o padrão P =* `"abacab"` *do Exemplo* 9.1. *A função de falha f (j) do algoritmo Knuth-Morris-Pratt (KMP) para P é mostrada na tabela a seguir*:

j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

O algoritmo de procura de padrões Knuth-Morris-Pratt, mostrado no Algoritmo 9.7, processa incrementalmente o texto T comparando-o ao padrão P. Cada vez que existir uma igualdade entre caracteres, incrementamos os índices correntes. Por outro lado, se não houver uma igualdade e anteriormente fizemos progresso em P, então consultamos a função de falha para determinar o novo índice em P, onde devemos continuar verificando P em T. Caso contrário, (ou seja, não há igualdade e estamos no início de P), incrementamos o índice de T (e mantemos o índice de P em seu início). Repetimos esse processo até encontrar uma ocorrência de P em T ou até o índice de T chegar a n, que é o comprimento de T (indicando que não achamos P em T).

A parte principal do algoritmo KMP é o laço ***enquanto***, que realiza a comparação entre um caracter de T e um caracter de P a cada iteração. Dependendo do resultado desta comparação, o algoritmo passa para o caracter seguinte em T e P, consultando a função de falha para um novo caracter candidato em P, ou reinicia o processo com o novo índice em T. A correção desse algoritmo segue da definição da função de falha. Quaisquer comparações que são evitadas são realmente desnecessárias, pois a função de falha garante que todas as comparações ignoradas são redundantes – elas envolveriam a comparação dos mesmos caracteres novamente.

Algoritmo KMPMatch (T, P):
  Entrada: as cadeias T (texto) com n caracteres e P (padrão) com m caracteres.
  Saída: o índice da primeira *substring* de T igual a P, ou uma indicação de que P não é uma *substring* de T.
  $f \leftarrow$ KMPFailureFunction(P) {construir a função de falha f para P}
  $i \leftarrow 0$
  $j \leftarrow 0$
  enquanto $i < n$ **faça**
    se $P[j] = T[i]$ **então**
      se $j = m - 1$ **então**
        retorne $i - m + 1$ {achamos!}
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
    senão se $j > 0$ {não achamos, mas avançamos em P} **então**
      $j \leftarrow f(j - 1)$ {j indexa logo após o prefixo de P que já achamos}
    senão
      $i \leftarrow i + 1$
  retorne "Não existe *substring* em T igual a P."

Algoritmo 9.7 Algoritmo KMP para procura de padrões.

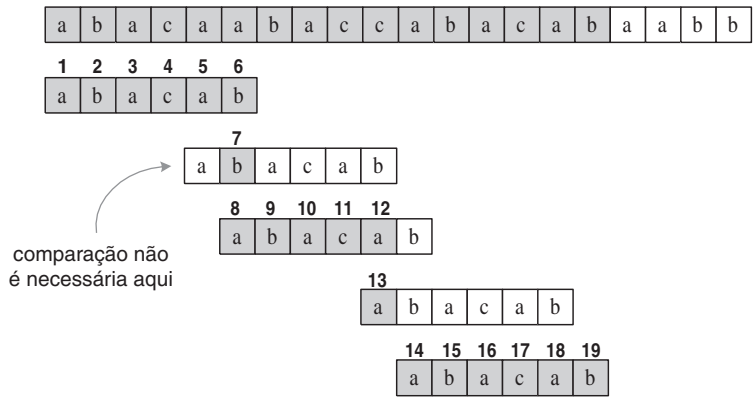


Figura 9.8 Ilustração do algoritmo de procura de padrões Knuth-Morris-Pratt. A função de falha f para este padrão é dada no Exemplo 9.2. O algoritmo realiza 19 comparações de caracteres, indicadas pela numeração.

Na Figura 9.8, ilustramos a execução do algoritmo KMP para procura de padrões nas mesmas cadeias de caracteres do Exemplo 9.1. Note o uso da função de falha para evitar refazer uma das comparações entre um caracter do padrão e um caracter do texto. Também note que o algoritmo realiza menos comparações do que o algoritmo de força bruta sobre as mesmas cadeias (Figura 9.2).

Desempenho

Excluindo a computação da função de falha, o tempo de execução do algoritmo Knuth-Morris-Pratt é claramente proporcional ao número de iterações do laço **enquanto**. Para uma análise, vamos definir $k = i - j$. Intuitivamente, k é o total de avanço do padrão P em relação a T. Note que em toda a execução do algoritmo temos $k \leq n$. Um dos três casos abaixo ocorre a cada iteração do laço:

- se $T[i] = P[j]$, então i aumenta em 1 e k não se altera, pois j também aumenta em 1;
- se $T[i] \neq P[j]$ e $j > 0$, então i não se altera e k aumenta em pelo menos 1, pois neste caso k muda de $i - j$ para $i - f(j - 1)$, que é maior do que $j - f(j - 1)$, que é positivo porque $f(j - 1) < j$;
- se $T[i] \neq P[j]$ e $j = 0$, então i aumenta em 1 e k aumenta em 1, pois j não se altera.

Assim, a cada iteração do laço, temos que i ou k aumentam em ao menos 1 (possivelmente ambos); portanto, o número total de iterações do laço **enquanto** no algoritmo KMP é de no máximo $2n$. Atingir este número, é claro, pressupõe que já computamos a função de falha para P.

Construção da função de falha de KMP

Para construir a função de falha, usamos o método mostrado no Algoritmo 9.9, que é um processo semelhante ao algoritmo KMPMatch. Comparamos o padrão consigo mesmo como no algoritmo KMP. Cada vez que temos dois caracteres iguais, fazemos $f(i) = j + 1$. Note que, como temos $i > j$ em toda a execução do algoritmo, $f(j - 1)$ está sempre definida quando precisamos usá-la.

Algoritmo KMPFailureFunction(P):
  Entrada: o padrão P, com m caracteres.
  Saída: a função de falha f para P, mapeando j para o comprimento do mais longo prefixo de P, que é um sufixo de $P[1..j]$.
  $i \leftarrow 1$
  $j \leftarrow 0$
  $f(0) \leftarrow 0$
  enquanto $i < m$ **faça**
    se $P[j] = P[i]$ **então**
      {já achamos $j + 1$ caracteres}
      $f(i) \leftarrow j + 1$
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
    senão se $j > 0$ **então**
      {j indexa logo após um prefixo de P que deve servir}
      $j \leftarrow f(j - 1)$
    senão
      {não achamos}
      $f(i) \leftarrow 0$
      $i \leftarrow i + 1$

Algoritmo 9.9 Computação da função de falha usada no algoritmo KMP de procura de padrões. Note como o algoritmo usa os valores previamente calculados da função de falha para calcular novos valores eficientemente.

O algoritmo KMPFailureFunction tem tempo $O(m)$. Sua análise é análoga à do algoritmo KMPMatch. Assim, temos:

Teorema 9.3: *o algoritmo Knuth-Morris-Pratt realiza a procura de padrões com uma cadeia de caracteres de comprimento n e um padrão de comprimento m em tempo* $O(n + m)$.

A análise do tempo de execução do algoritmo KMP pode parecer um pouco surpreendente, pois afirma que, em tempo proporcional ao necessário para ler as cadeias T e P separadamente, também podemos encontrar a primeira ocorrência de P em T. Também deve-se notar que o tempo de execução do algoritmo KMP não depende do tamanho do alfabeto.

9.2 *Tries*

Os algoritmos de procura de padrões apresentados na seção anterior aceleram a procura em um texto pré-processando o padrão (para computar a função de falha no algoritmo KMP ou a função last no algoritmo BM). Nesta seção, usaremos uma abordagem complementar, ou seja, apresentaremos algoritmos de procura que pré-processam o texto. Essa abordagem é adequada para aplicações em que uma série de consultas é realizada em um texto fixo, de forma que o custo inicial de pré-processar o texto é compensado pela aceleração das consultas seguintes (por exemplo, um *site* na Web que oferece consultas de *Hamlet* ou um mecanismo de busca que oferece páginas da Web sobre o tópico *Hamlet*).

Um ***trie*** é uma estrutura de dados baseada em árvore que serve para armazenar cadeias de caracteres e suportar uma rápida procura de padrões. A aplicação principal dos *tries* é na recuperação de informação. De fato, o nome "trie" vem da palavra "re*trie*val" (recuperação). Em uma aplicação de recuperação de informação, como a procura por uma seqüência de DNA em uma base de genomas, temos uma coleção S de cadeias de caracteres definidas usando-se o mesmo alfabeto.

As operações primárias de consulta suportadas por *tries* são a procura de padrões e a ***procura de prefixos***. A última operação envolve uma cadeia X e a determinação de todas as cadeias em S que tem X como prefixo.

9.2.1 *Tries* padrão

Seja S um conjunto de s cadeias de caracteres do alfabeto Σ de forma que nenhuma cadeia de S é um prefixo de outra cadeia. Um ***trie padrão*** para S é uma árvore ordenada T com as seguintes propriedades (veja a Figura 9.10):

- cada nodo de T, exceto a raiz, é rotulado com um caracter de Σ;
- a ordem dos filhos de um nodo interno em T é determinada por uma ordenação canônica do alfabeto Σ;
- T tem s nodos externos, cada um associado com uma cadeia de S de tal forma que a concatenação dos rótulos dos nodos no caminho da raiz até um nodo externo v de T resulte na cadeia de S associada com v.

Assim, um *trie T* representa as cadeias de *S* com os caminhos da raiz até os nodos externos de *T*. Note a importância de assumir que nenhuma cadeia de *S* seja prefixo de outra cadeia, pois isso garante que toda cadeia seja unicamente associada a um nodo externo de *T*. Podemos sempre satisfazer essa exigência adicionando um caracter especial que não se encontra no alfabeto Σ ao final de uma cadeia.

Um nodo interno em um *trie* padrão *T* pode ter entre 1 e *d* filhos, onde *d* é o tamanho do alfabeto. Existe um arco que vai da raiz *r* até um de seus filhos para cada primeiro caracter em alguma cadeia na coleção *S*. Adicionalmente, um caminho da raiz de *T* para um nodo interno *v* com profundidade *i* corresponde a um prefixo $X[0..i-1]$ com *i* caracteres de uma cadeia *X* em *S*. De fato, para cada caracter *c* que pode seguir o prefixo $X[0..i-1]$ em uma cadeia de *S*, existe um filho de *v* rotulado com o caracter *c*. Desta forma, um *trie* armazena concisamente os prefixos comuns que existem em um conjunto de cadeias.

Se existirem somente dois caracteres no alfabeto, então o *trie* é essencialmente uma árvore binária, embora alguns nodos internos possam ter somente um filho (ou seja, pode ser uma árvore binária imprópria). Em geral, se existirem *d* caracteres no alfabeto, então o *trie* será uma árvore múltipla onde cada nodo interno tem de 1 a *d* filhos. Além disso, é provável que existam vários nodos internos em *T* que tenham menos do que *d* filhos. Por exemplo, o *trie* mostrado na Figura 9.10 tem vários nodos internos com apenas um filho. Podemos implementar um *trie* com uma árvore que armazena caracteres em seus nodos.

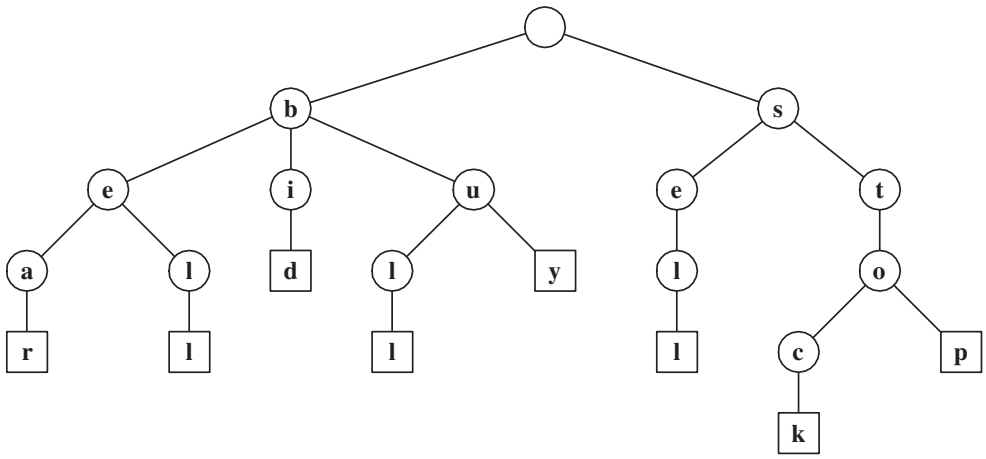
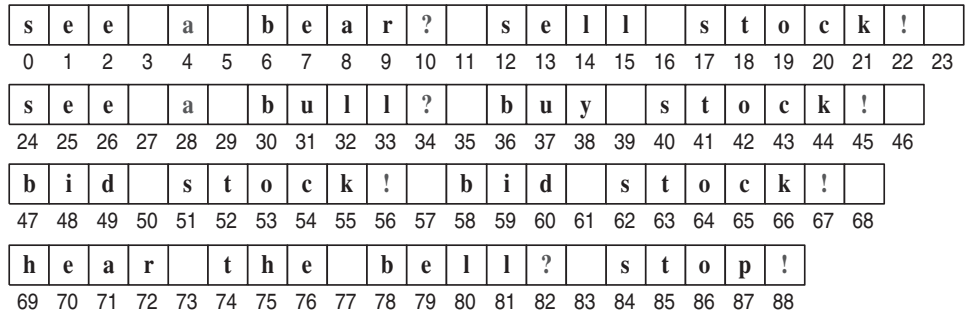


Figura 9.10 Um *trie* padrão para as cadeias {*bear*, *bell*, *bid*, *bull*, *buy*, *sell*, *stock*, *stop*}.

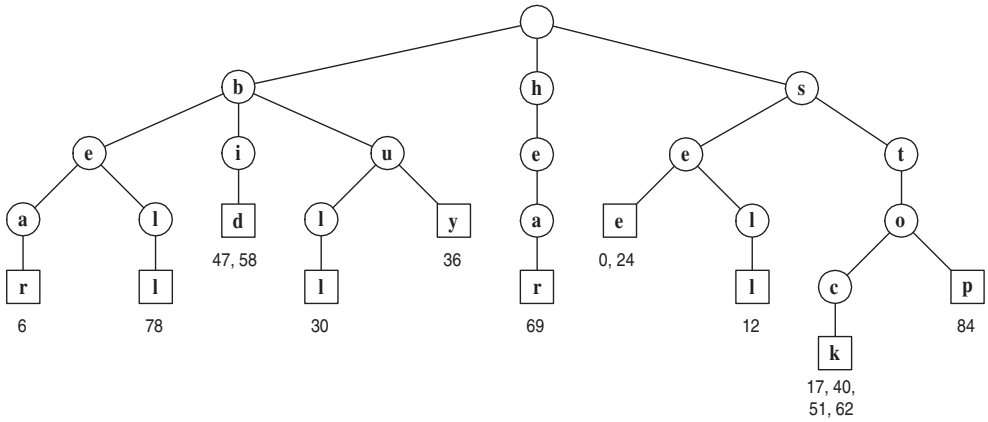
O Teorema a seguir descreve algumas propriedades estruturais importantes de um *trie* padrão:

Teorema 9.4: *um trie padrão que armazena uma coleção S de s cadeias de comprimento total n em um alfabeto de tamanho d apresenta as seguintes propriedades:*

- *todo nodo interno de T tem no máximo d filhos;*
- *T tem s nodos externos;*
- *a altura de T é igual ao comprimento da maior cadeia em S;*
- *o número de nodos em T é* $O(n)$.



(a)



(b)

Figura 9.11 Procura de palavras e procura de prefixos com um *trie* padrão: (a) texto a ser pesquisado; (b) *trie* padrão para as palavras no texto (com artigos e preposições, que também são conhecidos como ***stop words***, excluídos), com nodos externos aumentados para indicar as posições das palavras.

Desempenho

O pior caso para o número de nodos de um *trie* ocorre quando nenhum par de cadeias compartilha um prefixo, ou seja, exceto pela raiz, todos os nodos internos têm um filho.

Um *trie* T para um conjunto S de cadeias de caracteres pode ser usado para implementar um dicionário cujas chaves são as cadeias de S. Realizamos uma procura por uma cadeia X em T iniciando na raiz e seguindo os caracteres de X. Se esse caminho pode ser traçado e termina em um nodo externo, então X está no dicionário. Por exemplo, no *trie* da Figura 9.10, o traço do caminho para "*bull*" termina em um nodo externo. Se o caminho não pode ser traçado ou não termina em um nodo externo, então X não está no dicionário. No exemplo da Figura 9.10, o caminho para "*bet*" não pode ser traçado e o caminho para "*be*" termina em um nodo interno. Nenhuma destas palavras

está no dicionário. Note que, nesta implementação de um dicionário, caracteres isolados são comparados em vez de toda a cadeia (chave).

É fácil perceber que o tempo de execução para a procura de uma cadeia de comprimento m é $O(dm)$, onde d é o tamanho do alfabeto. De fato, visitamos no máximo $m + 1$ nodos de T e gastamos tempo $O(d)$ em cada nodo. Para alguns alfabetos, podemos melhorar o tempo gasto em um nodo para $O(1)$ ou $O(\log(d))$ usando um dicionário de caracteres implementado com um tabela *hash* ou outro tipo de tabela. Entretanto, já que d é uma constante na maioria das aplicações, podemos usar uma abordagem simples que use tempo $O(d)$ por nodo visitado.

A partir da discussão anterior, concluímos que podemos usar um *trie* para realizar um tipo especial de procura de padrões, chamado de ***procura de palavra***, no qual desejamos determinar se um dado padrão é exatamente igual a uma das palavras do texto (veja a Figura 9.11). A procura de palavras difere da procura de padrões comum, já que o padrão não pode ser igual a uma *substring* arbitrária do texto, mas a apenas uma de suas palavras. Usando um *trie*, a procura de palavras para um padrão de comprimento m custa tempo $O(dm)$, onde d é o tamanho do alfabeto, independentemente do tamanho do texto. Se o alfabeto tiver tamanho constante (caso das linguagens naturais e do DNA), uma consulta leva tempo $O(m)$, proporcional ao tamanho do padrão. Uma extensão simples deste esquema suporta procuras por prefixos. No entanto, ocorrências arbitrárias do padrão no texto (por exemplo, o padrão é um sufixo relativo a uma palavra ou abrange duas palavras) não podem ser eficientemente realizadas.

Para construir um *trie* padrão para um conjunto S de cadeias de caracteres, podemos usar um algoritmo incremental que insere uma cadeia de cada vez. Lembre-se da condição de que nenhuma cadeia de S seja um prefixo de outra cadeia de S. Para inserir uma cadeia X no *trie* corrente T, primeiramente tentamos traçar o caminho associado com X em T. Como X ainda não está em T e não é prefixo de nenhuma outra cadeia, pararemos de traçar o caminho em um nodo ***interno*** v de T antes de chegar ao final de X. Então criamos um novo caminho de nodos e descendentes de v para armazenar os caracteres restantes de X. O tempo para inserir X é $O(dm)$, onde m é o comprimento de X e d é o tamanho do alfabeto. Assim, construir um *trie* completo para o conjunto S leva tempo $O(dn)$, onde n é o comprimento total das cadeias de caracteres em S.

Existe uma potencial ineficiência de espaço no *trie* padrão que provocou o desenvolvimento do ***trie comprimido***, que é também conhecido (por razões históricas) como ***trie Patricia***. A saber, existem potencialmente muitos nodos no *trie* padrão que têm apenas um filho, e a existência desses nodos é um desperdício, pois implica que o número total de nodos na árvore poderia ser maior do que o número de palavras no texto correspondente.

Discutiremos na próxima subseção a estrutura de dados para o *trie* comprimido.

9.2.2 *Tries* comprimidos

Um ***trie comprimido*** é similar a um *trie* padrão, mas garante que cada nodo interno no *trie* tenha pelo menos dois filhos. Ele impõe essa regra comprimindo cadeias de nodos com apenas um filho em nodos isolados (veja a Figura 9.12). Seja T um *trie* padrão. Dizemos que um nodo v de T é ***redundante*** se v tem um filho e não for a raiz. Por exemplo, o *trie* da Figura 9.10 tem oito nodos redundantes. Vamos agora mostrar que uma cadeia de $k \geq 2$ nodos,

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k),$$

é ***redundante*** se

- v_i for redundante para $i = 1... k - 1$;
- v_0 e v_k não forem redundantes.

Podemos transformar T em um *trie* comprimido substituindo cada cadeia redundante $(v_0,v_1),...,(v_{k-1},v_k)$ com $k \geq 2$ arestas em uma única aresta (v_0,v_k), renomeando v_k com a concatenação dos rótulos dos nodos $v_1,...,v_k$.

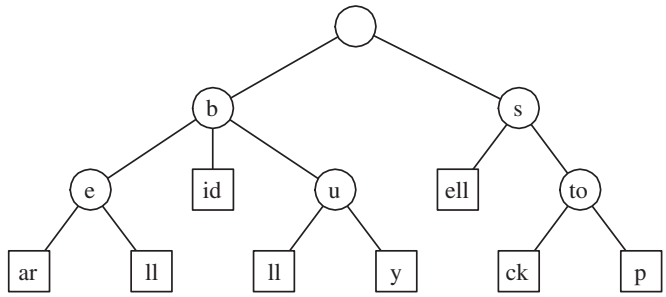


Figura 9.12 *Trie* comprimido para as cadeias {*bear*, *bell*, *bid*, *bull*, *buy*, *sell*, *stock*, *stop*}. Compare-o com o *trie* padrão mostrado na Figura 9.10.

Assim, os nodos em um *trie* comprimido são rotulados com cadeias de caracteres, que são *substrings* das cadeias na coleção, em vez de caracteres isolados. A vantagem de um *trie* comprimido sobre um *trie* padrão é que o número de nodos de um *trie* comprimido é proporcional ao número de cadeias de caracteres, e não a seu comprimento total, como mostrado no Teorema a seguir (compare com o Teorema 9.4).

Teorema 9.5: *um trie comprimido que armazena uma coleção S de s cadeias de caracteres de um alfabeto de tamanho d tem as seguintes propriedades:*

- *todo nodo interno de T tem ao menos dois e no máximo d filhos;*
- *T tem s nodos externos;*
- *o número de nodos de T é* $O(s)$.

O leitor atento pode se perguntar se a compressão dos caminhos traz alguma vantagem significativa, ela é compensada por um crescimento dos rótulos dos nodos. De fato, um *trie* comprimido é realmente vantajoso somente quando for usado como uma estrutura ***auxiliar*** de índice para uma coleção de cadeias de caracteres já armazenadas em uma estrutura primária, e quando não há necessidade de armazenar realmente todos os caracteres das cadeias da coleção. No entanto, com esta estrutura auxiliar, o *trie* comprimido é bastante eficiente.

Suponha, por exemplo, que a coleção S de cadeias de caracteres é um arranjo de cadeias $S[0]$, $S[1]$,..., $S[s - 1]$. Em vez de armazenar explicitamente o rótulo X de um nodo, ele é representado implicitamente por uma tripla de inteiros (i,j,k) de forma que $X = S[i][j..k]$; ou seja, X é a *substring* de $S[i]$ que consiste nos caracteres da j-ésima até a k-ésima posição inclusive. (Veja o exemplo da Figura 9.13. Também compare com o *trie* padrão da Figura 9.11.)

Esse esquema adicional de compressão permite-nos reduzir o total de espaço de $O(n)$ (*trie* padrão) para $O(s)$ (*trie* comprimido), onde n é o comprimento total das ca-

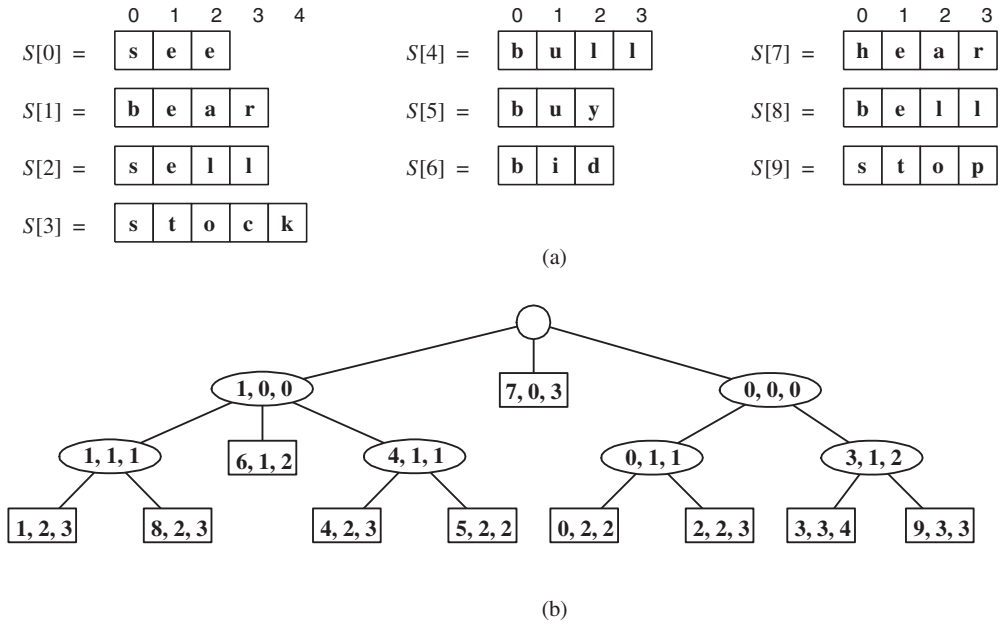


Figura 9.13 (a) Coleção *S* de cadeias de caracteres armazenadas em um arranjo. (b) Representação compacta do *trie* comprimido para *S*.

deias em *S*, e *s* é o número de cadeias em *S*. É claro que ainda temos de armazenar as cadeias em *S*, mas ainda assim reduzimos a necessidade de espaço para o *trie*. Na próxima seção, apresentaremos uma aplicação em que a coleção de cadeias de caracteres também pode ser armazenada de forma compacta.

9.2.3 *Tries* de sufixos

Uma das aplicações primordiais dos *tries* é para o caso em que as cadeias de caracteres da coleção *S* são todas os sufixos da cadeia *X*. Um *trie* como esse é chamado de ***trie de sufixos***, (também conhecido como ***árvore de sufixos*** ou ***árvore de posições***) de uma cadeia *X*. Por exemplo, a Figura 9.14a mostra o *trie* de sufixos para os oito sufixos da cadeia de caracteres `"minimize"`.

Para um *trie* de sufixos, a representação compacta apresentada na seção anterior pode ser ainda mais simplificada. Isso quer dizer que o rótulo de cada vértice é um par (i,j) indicando a cadeia $X[i..j]$ (veja a Figura 9.14b). Para satisfazer a regra de que nenhum sufixo de *X* seja prefixo de outro sufixo, podemos adicionar um caracter especial, denotado por \$ que não faça parte do alfabeto original Σ ao final de *X* (e assim a todos os sufixos). Ou seja, se a cadeia *X* tiver comprimento *n*, construímos um *trie* para o conjunto de *n* cadeias $X[i..n-1]\$$, para $i = 0,..., n-1$.

Economia de espaço

Usar um *trie* de sufixos permite-nos economizar espaço em relação a um *trie* padrão, usando várias técnicas de compressão, incluindo as usadas para um *trie* comprimido.

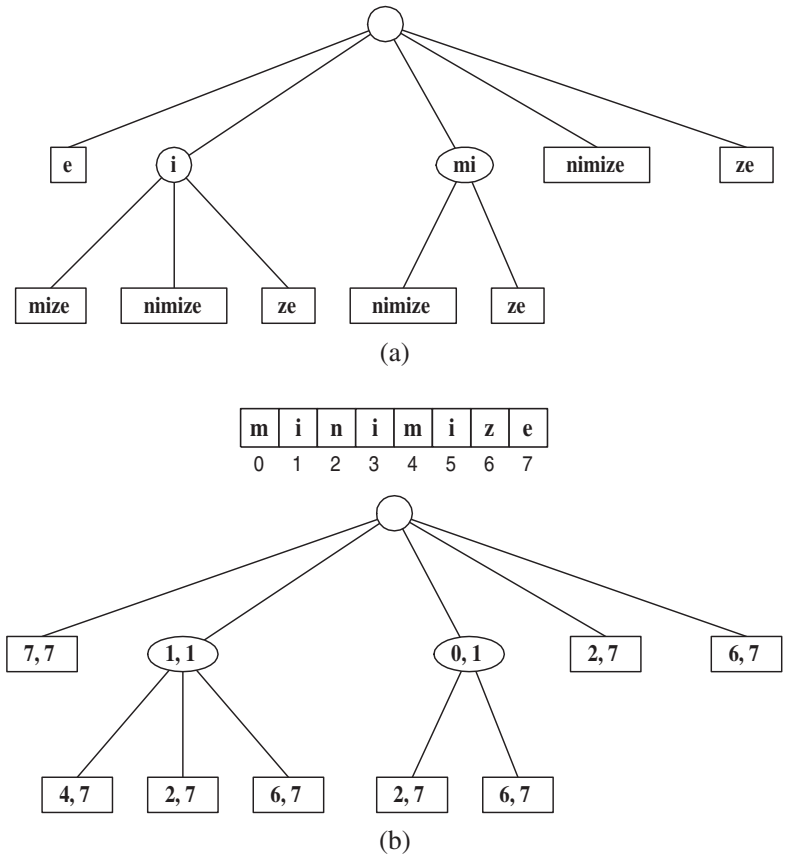


Figura 9.14 (a) *Trie* de sufixos T para a cadeia X = `"minimize"`. (b) Representação compacta de T, onde o par (i,j) denota $X[i..j]$.

A vantagem da representação compacta torna-se aparente para os *tries* de sufixos. Já que o comprimento total dos sufixos de uma cadeia X de comprimento n é

$$1+2+\cdots+n=\frac{n(n+1)}{2},$$

o armazenamento explícito de todos os sufixos de X ocuparia espaço $O(n^2)$. Mesmo assim, o *trie* de sufixos representa implicitamente essas cadeias em espaço $O(n)$, como formalizado no Teorema a seguir.

Teorema 9.6: *a representação compacta de um trie de sufixos T para uma cadeia de caracteres X de comprimento n utiliza espaço* $O(n)$.

Construção

Podemos construir o *trie* de sufixos para uma cadeia de comprimento n com um algoritmo incremental como o da Seção 9.2.1. Essa construção leva tempo $O(dn^2)$, pois o comprimento total dos sufixos é quadrático em n. Entretanto, o *trie* de sufixos (compacto) para uma cadeia de comprimento n pode ser construído em tempo $O(n)$ com um algoritmo especializado diferente do usado para *tries* padrão. Esse algoritmo de construção em tempo linear, entretanto, é bastante complexo, e não será relatado aqui. Con-

tudo, podemos tirar vantagem da existência desse algoritmo rápido de construção quando desejarmos usar um *trie* de sufixos para resolver outros problemas.

Usando um *trie* de sufixos

O *trie* de sufixos T para uma cadeia de caracteres X pode ser usado eficientemente para realizar procura de padrões no texto X. Ou seja, podemos determinar se um padrão P é uma *substring* de X tentando traçar um caminho associado a P em T. P é uma *substring* de X se e somente se um tal caminho puder ser traçado. Os detalhes do algoritmo de procura de padrões são dados no Algoritmo 9.15, que pressupõe a seguinte propriedade adicional nos rótulos dos nodos na representação compacta do *trie* de sufixos:

> Se o nodo v tiver rótulo (i,j) e Y for a cadeia de comprimento y associada com o caminho da raiz até v (inclusive), então $X[j - y + 1..j] = Y$.

Essa propriedade assegura que podemos calcular facilmente o índice de início do padrão no texto quando ele for encontrado.

```
Algoritmo sufixTrieMatch(T, P):
    Entrada: o trie de sufixos comprimido T para o texto X e o padrão P.
    Saída: o índice de início de uma substring de X igual a P ou uma indicação
      de que P não é uma substring de X.
    p ← P. length() {comprimento do sufixo do padrão a ser encontrado}
    j ← 0 {início do sufixo do padrão a ser encontrado}
    v ← T. root()
    repete
      f ← verdadeiro {indicador de que nenhum filho foi processado com sucesso}
      para cada filho w de v faça
        i ← start(w)
        se P[j] = T [i] então
          {processa o filho w}
          x ← end(w) – i + 1
          se p ≤ x então
            {sufixo é mais curto ou do mesmo tamanho do que o rótulo do nodo}
            se P[j..j + p – 1] = X [i..i + p – 1] então
              retorne i – j {achamos}
            senão
              retorne "P não é uma substring de X"
          senão
            {sufixo é mais longo do que o rótulo do nodo}
            se P[j..j + x – 1] – X [i..i + x – 1] então
                  p ← p – x {atualiza o comprimento do sufixo}
                  j ← j + x {atualiza índice de início do sufixo}
                  v ← w
                  f ← falso
                  quebra encerra o laço para
    até f ou T.IsExternal(v)
    retorne "P não é uma substring de X"
```

Algoritmo 9.15 Procura de padrões com um *trie* de sufixos. Denotamos o rótulo de um nodo v com (start (v), end(v)), ou seja, o par de índices especificando a *substring* do texto associada a v.

Propriedades do *trie* de sufixos

A correção do algoritmo suffixTrieMatch resulta do fato que procuramos no *trie T* encontrando um caracter do padrão *P* de cada vez até que um dos seguintes eventos aconteça:

- achamos o padrão *P;*
- confirmamos que não podemos encontrar o padrão *P* (quando o laço **para** termina sem uma **quebra**);
- ainda temos caracteres de *P* a serem encontrados após processarmos um nodo externo.

Seja m o comprimento do padrão *P,* e d o tamanho do alfabeto. Para determinar o tempo de execução do algoritmo suffixTrieMatch, fazemos as observações seguintes:

- processamos no máximo $m + 1$ nodos do *trie*;
- cada nodo processado tem no máximo d filhos;
- a cada nodo v processado, realizamos no máximo uma comparação de caracter para cada filho w de v para determinar qual filho de v deve ser processado em seguida (o que possivelmente pode ser acelerado usando um dicionário como índice dos filhos de v);
- realizamos no máximo m comparações de caracteres nos nodos processados;
- gastamos tempo $O(1)$ para cada comparação de caracter.

Desempenho

Concluímos que o algoritmo suffixTrieMatch realiza procura de padrões em tempo $O(dm)$ (e poderia ser realizado ainda mais rapidamente se usássemos um dicionário para indexar os filhos de nodos no *trie* de sufixos). Note que o tempo de execução não depende do comprimento do texto X. Além disso, o tempo de execução é linear no comprimento do padrão, ou seja, $O(m)$ para um alfabeto de tamanho constante. Portanto, *tries* de sufixos são adequados para aplicações repetitivas de procura de padrões, em que uma série de procuras é realizada em um texto fixo.

Resumimos os resultados desta seção no teorema a seguir.

Teorema 9.7: *seja X um texto com n caracteres de um alfabeto de tamanho d. Podemos realizar procura de padrões em X em tempo $O(dm)$, onde m é o comprimento do padrão, com o trie de sufixos de X, que usa espaço $O(n)$ e pode ser construído em tempo $O(dn)$.*

Exploraremos outra aplicação dos *tries* na próxima subseção.

9.2.4 Mecanismos de busca

A World Wide Web (WWW) contém uma imensa coleção de documentos textuais (páginas). As informações sobre essas páginas são coletadas por programas chamados ***Web crawlers***, que armazenam essas informações em bancos de dados especiais de dicionários. Um ***mecanismo de busca*** na Web permite que usuários recuperem informações relevantes desses bancos de dados, identificando páginas relevantes na Web que contenham determinadas palavras-chave. Nesta seção, apresentaremos um modelo simplificado de um mecanismo de busca.

Arquivos invertidos

A informação básica armazenada por um mecanismo de busca é um dicionário chamado de ***índice invertido*** ou ***arquivo invertido***, que armazena pares de chave e valor (w, L), onde w é uma palavra e L é uma coleção de páginas contendo a palavra w. As chaves (palavras) no dicionário são chamadas de ***termos de índice*** e deveriam ser o maior conjunto de itens de vocabulário e nomes próprios possível. Os elementos deste dicionário são chamados de ***listas de ocorrências*** e devem cobrir o maior número de páginas da Web.

Podemos implementar eficientemente um índice invertido com uma estrutura de dados que consista em:

- um arranjo armazenando as listas de ocorrências dos termos (sem ordenação);
- um *trie* comprimido para o conjunto de termos de índice, onde cada nodo externo armazena o índice da lista de ocorrência do termo associado.

A razão para armazenarmos as listas de ocorrências fora do *trie* é manter o tamanho do *trie* pequeno o bastante para caber em uma memória interna. Em vez disso, por causa de seu grande tamanho total, as listas de ocorrências são armazenadas em disco.

Com nossa estrutura de dados, uma consulta por uma única palavra-chave é similar a uma procura por uma palavra em um texto (veja a Seção 9.2.1). Ou seja, achamos a palavra-chave no *trie* e retornamos a lista de ocorrências associada.

Quando múltiplas palavras-chave são dadas e a saída desejada é composta pelas páginas contendo **todas** as palavras, recuperamos a lista de ocorrência de cada palavra-chave usando o *trie* e retornamos sua intersecção. Para facilitar o cálculo da intersecção, cada lista de ocorrências poderia ser implementada como uma seqüência ordenada por endereço ou com um dicionário (veja, por exemplo, a computação genérica de junção discutida na Seção 4.2).

Além da tarefa básica de retornar a lista de páginas contendo as palavras-chave dadas, mecanismos de busca fornecem um importante serviço adicional ***classificando*** as páginas retornadas por relevância. Projetar algoritmos de classificação rápidos e precisos para mecanismos de busca é um desafio para os pesquisadores em computação e para as empresas de comércio eletrônico.

9.3 Compressão de textos

Nesta seção, consideramos outra aplicação do processamento de textos, a ***compressão de texto***. Neste problema, temos uma cadeia de caracteres X definida sobre um dado alfabeto como ASCII ou Unicode e desejamos codificar X de forma eficiente em uma pequena cadeia binária Y (usando apenas os caracteres 0 e 1). A compressão de textos é útil em qualquer situação em que nos comunicamos através de um canal de baixa capacidade de transmissão, tal como um modem ou conexão infravermelha, e desejamos minimizar o tempo necessário para transmitir o texto. A compressão de texto também é útil para armazenar coleções de documentos mais eficientemente, permitindo que um meio de armazenamento de capacidade fixa comporte o maior número de documentos quanto possível.

O método para compressão de textos explorado nesta seção é o ***código de Huffman***. Esquemas padrão de codificação, tais como ASCII e Unicode, usam cadeias binárias de comprimento fixo para codificar caracteres (com 7 *bits* no caso de ASCII e 16 no Unico-

de). Um código de Huffman, por outro lado, usa uma codificação de comprimento variável otimizada para o caracter X. A otimização é baseada no uso de ***freqüências*** de caracteres, onde temos, para cada caracter c, uma contagem $f(c)$ do número de vezes em que c aparece na cadeia X. O código de Huffman economiza espaço em relação a um código de comprimento fixo usando cadeias curtas para codificar caracteres com alta freqüência e cadeias mais longas para caracteres com baixa freqüência.

Para codificar a cadeia X, convertemos cada caracter em X, de sua representação em comprimento fixo, para uma representação de comprimento variável e concatenamos todas essas representações para produzir a codificação Y de X. Para evitar ambigüidades, insistimos que nenhum código em nossa codificação seja um prefixo de outro código. Uma codificação desse tipo é chamada de ***codificação de prefixos*** e ela simplifica a decodificação de Y para obter X novamente (veja a Figura 9.16). Mesmo com essa restrição, a economia produzida por uma codificação de prefixos de comprimento variável pode ser significativa, particularmente se existir uma grande variação nas freqüências dos caracteres (como é o caso do texto em linguagem natural em quase qualquer idioma).

O algoritmo de Huffman, para produzir uma codificação de prefixos de comprimento variável que seja ótima para X, é baseado na construção de uma árvore binária T que representa a codificação. Cada nodo em T, exceto a raiz, representa um *bit* no código de um caracter, representando cada filho esquerdo um "0" e cada filho direito, "1". Cada nodo externo v é associado a um caracter específico, e o código para esse caracter é definido pela seqüência de *bits* associada com os nodos no caminho da raiz de T até v (veja a Figura 9.16). Cada nodo externo v tem uma ***freqüência*** $f(v)$, que é simplesmente a freqüência em X do caracter associado com v. Adicionalmente, damos a cada nodo interno v em T uma freqüência $f(v)$, que é a soma das freqüências de todos os nodos externos na subárvore com raiz v.

(a)

Caracter		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Freqüência	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

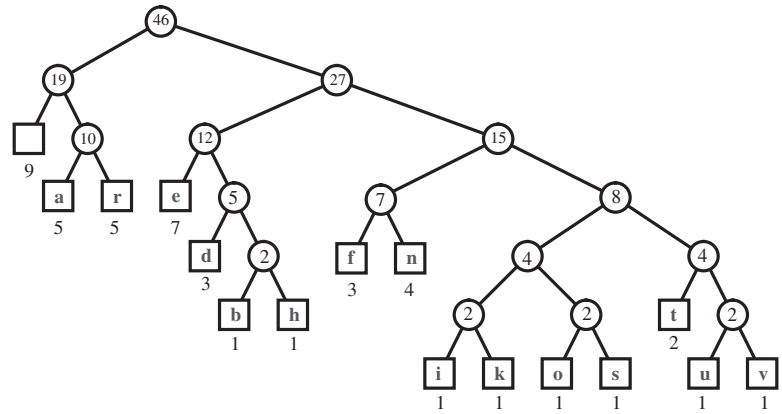


Figura 9.16 Ilustração do código de Huffman para a cadeia de caracteres X = `"a fast runner need never be afraid of the dark"`: (a) freqüência de cada caracter de X; (b) árvore de Huffman T para a cadeia X. O código para um caracter c é obtido traçando o caminho da raiz de T até o nodo externo onde c está armazenado e associando um filho à esquerda com 0 e um filho à direita com 1. Por exemplo, o código para "a" é 010, e o código para "f" é 1100.

9.3.1 O algoritmo de codificação de Huffman

O algoritmo de codificação de Huffman começa com cada um dos d diferentes caracteres da cadeia X a serem codificados sendo o nodo da raiz de uma árvore binária de um único nodo. O algoritmo procede com uma série de rodadas: em cada rodada, o algoritmo unifica as duas árvores binárias com menor freqüência em uma única árvore binária. Essa operação é repetida até que apenas uma árvore exista (veja o Algoritmo 9.17).

Cada iteração do laço **enquanto** no algoritmo de Huffman pode ser implementada em tempo $O(\log(d))$ usando-se uma fila de prioridades Q representada com um *heap*. Além disso, cada iteração retira dois elementos de Q e adiciona um, processo que se repete $d - 1$ vezes antes de exatamente um nodo estar em Q. Assim, este algoritmo é executado em tempo $O(n + d \log(d))$. Embora uma justificativa completa da correção deste algoritmo esteja além de nosso escopo, mostramos que ele é intuitivo através de uma idéia simples: qualquer codificação ótima pode ser convertida em uma codificação ótima em que os códigos para os dois caracteres de menor freqüência, a e b, diferem apenas em seu último *bit*. A repetição do argumento para uma cadeia com a e b substituídos por um caracter c fornece o teorema a seguir:

Teorema 9.8: *o algoritmo de Huffman constrói uma codificação por prefixos ótima para uma cadeia de caracteres de comprimento n com d caracteres distintos em tempo* $O(n + d \log(d))$.

Algoritmo Huffman(X):
  Entrada: a cadeia de caracteres X de comprimento n com d caracteres distintos.
  Saída: uma árvore com a codificação de X.
  Calcule a freqüência $f(c)$ de todo caracter c em X.
  Inicialize uma fila de prioridade Q.
  para cada character c in X **faça**
    Crie uma árvore binária T de um único nodo armazenando c.
    Insira T em Q com chave $f(c)$.
  enquanto Q.size() > 1 **faça**
    $f_1 \leftarrow Q$.minKey()
    $T_1 \leftarrow Q$.removeMin()
    $f_2 \leftarrow Q$.minKey()
    $T_2 \leftarrow Q$.removeMin()
    Crie uma nova árvore binária T com subárvore esquerda T_1 e subárvore direita T_2.
    Insira T em Q com chave $f_1 + f_2$.
  retorne tree Q.removeMin()

Algoritmo 9.17 Algoritmo de codificação de Huffman.

9.3.2 O método guloso revisitado

O algoritmo de Huffman para construção de uma codificação ótima é um exemplo de aplicação de um padrão de projeto chamado de ***método guloso***. Recordamos da Seção 5.1 que este padrão de projeto é aplicado em problemas de otimização em que tentamos construir alguma estrutura enquanto minimizamos ou maximizamos alguma propriedade dessa estrutura.

De fato, o algoritmo de Huffman segue a fórmula geral para o padrão dos algoritmos gulosos. Para resolver um dado problema de otimização usando o método guloso, fazemos uma seqüência de escolhas. A seqüência inicia com alguma condição inicial bem conhecida e calcula o custo para tal condição. O padrão então exige que iterativamente façamos escolhas adicionais, identificando a decisão que atinge o melhor custo dentre todas as escolhas que são possíveis no momento. Essa abordagem nem sempre resulta na solução ótima, mas quando usada no algoritmo de Huffman, ela resulta na melhor codificação possível.

Essa otimalidade para o algoritmo de Huffman se deve ao fato de que o problema da codificação ótima por prefixos possui a propriedade da ***escolha gulosa***. Essa é a propriedade que determina que uma solução global ótima pode ser atingida através de uma seqüência de soluções locais ótimas (ou seja, escolhas que são as melhores dentre as disponíveis a cada momento), começando por uma condição inicial bem definida. O problema de computar uma codificação de prefixos ótima de comprimento variável é apenas um exemplo de um problema que apresenta a propriedade da escolha gulosa.

9.4 Teste de similaridade de textos

Um problema comum em processamento de texto, presente na genética e na engenharia de *software*, é testar a similaridade entre duas cadeias de caracteres. Em uma aplicação em genética, as duas cadeias de caracteres poderiam corresponder a duas seqüências de DNA, que poderiam, por exemplo, vir de dois indivíduos que consideraríamos geneticamente relacionados se eles tivessem uma longa subseqüência comum em suas seqüências de DNA. Em uma aplicação de engenharia de *software*, as duas cadeias de caracteres poderiam vir de duas versões do mesmo programa, e poderíamos desejar determinar quais mudanças foram efetuadas de uma versão para a outra. Além disso, sistemas de coleta de informação, chamados de ***spiders*** ou ***crawlers,*** usados por mecanismos de busca, devem poder distinguir páginas semelhantes para evitar buscas desnecessárias. De fato, determinar a similaridade entre duas cadeias de caracteres é considerado uma operação tão comum que os sistemas operacionais Unix e Linux incluem com um programa, chamado de `diff`, para comparar arquivos de texto.

9.4.1 O problema da maior subseqüência comum

Existem várias formas diferentes de definir a similaridade entre duas cadeias de caracteres. Mesmo assim, podemos abstrair uma versão simples porém comum deste problema usando cadeias de caracteres e suas subseqüências. Dada uma cadeia X de comprimento n, uma ***subseqüência*** de X é qualquer cadeia da forma

$$X[i_1]X[i_2] \cdots X[i_k], \quad i_j < i_{j+1} \text{ para } j = 1, \ldots, k-1;$$

ou seja, uma seqüência de caracteres não são necessariamente contíguos, mas que são retirados de X em ordem. Por exemplo, a cadeia *AAAG* é uma subseqüência da cadeia *CGATAATTGAGA*. Note que o conceito de subseqüência de uma cadeia é diferente do conceito de uma *substring* de uma cadeia, definido na Seção 9.1.1.

Definição do problema

O problema específico de similaridade de textos que abordaremos aqui é o problema da ***maior subseqüência comum*** (LCS*). Neste problema, temos duas cadeias de caracteres, X de tamanho n e Y de tamanho m, definidas sobre algum alfabeto, e desejamos encontrar a cadeia mais longa S, que é uma subseqüência de X e Y.

Uma maneira de resolver o problema da maior subseqüência comum é enumerar todas as subseqüências de X e escolher a mais longa que também seja uma subseqüência de Y. Já que cada caracter de X ou está ou não está na subseqüência, existem potencialmente 2^n subseqüências diferentes de X, cada uma das quais requerendo tempo $O(m)$ para determinar se é uma subseqüência de Y. Assim, esta abordagem de força bruta fornece um algoritmo exponencial executado em tempo $O(2^n m)$, que é muito ineficiente. Nesta seção, discutiremos como usar a ***programação dinâmica*** (Seção 5.3) para resolver o problema da maior subseqüência comum muito mais rapidamente.

9.4.2 Aplicação da programação dinâmica no problema da LCS

Podemos resolver o problema da maior subseqüência comum muito mais depressa do que em tempo exponencial usando a técnica da programação dinâmica. Como mencionado na Seção 5.3, um dos componentes-chave da programação dinâmica é a definição de subproblemas simples que satisfazem as propriedades de otimização e intersecção dos subproblemas.

Lembre que no problema da LCS recebemos duas cadeias de caracteres X e Y de comprimentos n e m, respectivamente, e que desejamos encontrar a mais longa cadeia S, que é uma subseqüência de X e de Y. Como X e Y são cadeias de caracteres, temos um conjunto natural de índices com o qual se definem os subproblemas – índices para as cadeias X e Y. Vamos, portanto, definir um subproblema, como calcular o comprimento da mais longa subseqüência comum de $X[0..i]$ e $Y[0..j]$, denotada por $L[i,j]$.

Essa definição nos permite reescrever $L[i,j]$ em função das soluções ótimas de subproblemas. Consideramos os dois casos a seguir (veja Figura 9.18).

Caso 1: $X[i] = Y[j]$. Seja $c = X[i] = Y[j]$. Afirmamos que a maior subseqüência comum de $X[0..i]$ e $Y[0..j]$ termina com c. Para justificar essa afirmativa, vamos supor que ela não é verdadeira. Tem de haver alguma maior subseqüência comum $X[i_1]X[i_2]\ldots X[i_k] = Y[j_1]Y[j_2]\ldots Y[j_k]$ mais longa. Se $X[i_k] = c$ ou $Y[j_k] = c$, então temos a mesma seqüência fazendo $i_k = i$ e $j_k = j$. Alternativamente, se $X[i_k] \neq c$, então podemos obter uma maior subseqüência comum adicionando c ao final. Assim, uma maior subseqüência comum de $X[0..i]$ e $Y[0..j]$ termina com $c = X[i] = Y[j]$. Portanto, fazemos

$$L[i, j] = L[i-1, j-1] + 1 \quad \text{se } X[i] = Y[j]. \tag{9.1}$$

Caso 2: $X[i] \neq Y[j]$. Neste caso, não podemos ter uma subseqüência comum que inclui $X[i]$ e $Y[j]$. Ou seja, podemos ter uma subseqüência comum terminando com $X[i]$, $Y[j]$ ou possivelmente nenhum dos dois, mas certamente não ambos. Portanto, fazemos

$$L[i, j] = \max\{L[i-1, j], L[i, j-1]\} \quad \text{se } X[i] \neq Y[j]. \tag{9.2}$$

* N. de R. Em inglês, *longest common subsequence*.

Para que as Equações 9.1 e 9.2 façam sentido nos casos limite em que $i = 0$ ou $j = 0$, fazemos $L[i, -1] = 0$ para $i = -1,0,1,..., n - 1$ e $L[-1,j] = 0$ para $j = -1,0,1,..., m - 1$.

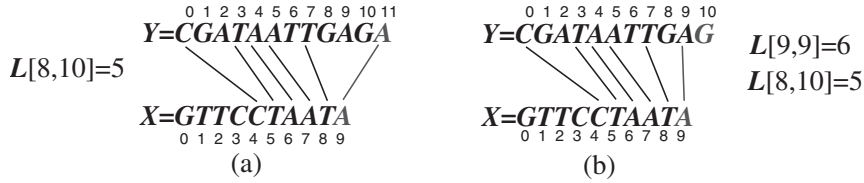


Figura 9.18 Os dois casos na definição de $L[i,j]$: (a) $X[i] = Y[j]$; (b) $X[i] \neq Y[j]$.

O algoritmo LCS

A definição anterior para $L[i,j]$ satisfaz a otimização dos subproblemas, pois não podemos ter uma maior subseqüência comum sem também haver uma maior subseqüência comum para os subproblemas. A definição também usa a intersecção de subproblemas, pois a solução de um subproblema $L[i,j]$ pode ser usada em vários outros problemas (a saber, os problemas $L[i + 1,j]$, $L[i,j + 1]$ e $L[i + 1,j + 1]$).

Transformar essa definição de $L[i,j]$ em um algoritmo é realmente bastante simples. Inicializamos um arranjo L de dimensões $(n + 1) \times (m + 1)$ com os casos limite em que $i = 0$ ou $j = 0$. Ou seja, inicializamos $L[i, -1] = 0$ para $i = -1,0,1,..., n - 1$ e $L[-1,j] = 0$ para $j = - 1,0,1,..., m - 1$. (Este é um leve abuso de notação, pois, na realidade, deveríamos indexar as linhas e colunas de L começando em 0.) Então, construímos os valores de L iterativamente até que obtemos $L[n - 1, m - 1]$, o comprimento da maior subseqüência comum entre X e Y. Fornecemos uma descrição em pseudocódigo da forma como esta abordagem resulta em uma solução com programação dinâmica para o problema da maior subseqüência comum (LCS) no Algoritmo 9.19. Note que o algoritmo armazena apenas os valores de $L[i,j]$.

Algoritmo LCS(X, Y):
  Entrada: as cadeias X e Y com n e m elementos, respectivamente.
  Saída: para $i = 0,1,..., n - 1$ e $j = 0,1,..., m - 1$, o comprimento $L[i..j]$
    da cadeia mais longa, que é uma subseqüência tanto de $X[0..i] = x_0, x_1... x_i$
    quanto de $Y[0..j] = y_0, y_1... y_j$.}
  para $i \leftarrow -1$ até $n - 1$ **faça**
    $L[i, 1] \leftarrow 0$
  para $j \leftarrow 0$ até $m - 1$ **faça**
    $L[-1, j] \leftarrow 0$
  para $i \leftarrow 0$ até $n - 1$ **faça**
    para $j \leftarrow 0$ até $m - 1$ **faça**
      se $X [i] = Y [j]$ **então**
        $L[i, j] \leftarrow L[i - 1, j - 1] + 1$
      senão
        $L[i, j] \leftarrow \max\{L[i - 1, j], L[i, j - 1]\}$
  retorne L

Algoritmo 9.19 Algoritmo de programação dinâmica para o problema da LCS.

Desempenho

O tempo de execução do Algoritmo 9.19 é fácil de analisar, pois é dominado por dois laços **para** aninhados: o externo iterando n vezes e o interno iterando m vezes. Já que o comando **se** e a atribuição dentro do laço requerem $O(1)$ operações primitivas, este algoritmo é executado em tempo $O(mn)$. Assim, a técnica da programação dinâmica pode ser aplicada ao problema da maior subseqüência comum para melhorar significativamente a solução de tempo exponencial baseada em força bruta.

O algoritmo LCS (9.19) computa o comprimento da maior subseqüência comum (armazenada em $L[n-1,m-1]$), mas não a subseqüência em si. Como é mostrado no próximo teorema, um simples passo de pós-processamento pode extrair a maior subseqüência comum do arranjo L retornado pelo algoritmo.

Teorema 9.9: *dada uma cadeia X de n caracteres e uma cadeia Y de m caracteres, podemos encontrar a maior subseqüência comum entre X e Y em tempo O(mn).*

Prova: o algoritmo LCS computa $L[n-1, m-1]$, o ***comprimento*** da maior subseqüência comum, em tempo $O(mn)$. Dada a tabela de valores $L[i,j]$, construir a maior subseqüência comum é imediato. Um método é começar de $L[n,m]$ e retornar através da tabela, reconstruindo uma maior subseqüência comum do fim para o início. Em qualquer posição $L[i,j]$, podemos determinar se $X[i] = Y[j]$. Em caso afirmativo, podemos usar $X[i]$ como o próximo caracter da subseqüência (notando que $X[i]$ fica ***antes*** de qualquer caracter que tenhamos achado anteriormente), movendo-nos para $L[i-1,j-1]$. Se $X[i] \neq Y[j]$, então podemos nos mover para o maior dentre $L[i,j-1]$ e $L[i-1,j]$ (veja a Figura 9.20). Paramos quando atingimos uma célula da borda (com $i = -1$ ou $j = -1$). Este método constrói uma maior subseqüência comum em tempo adicional $O(m+n)$. ■

L	–1	0	1	2	3	4	5	6	7	8	9	10	11
–1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

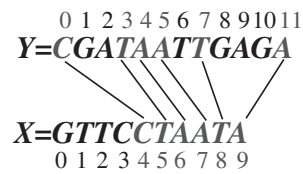


Figura 9.20 Ilustração do algoritmo para construir a maior subseqüência comum a partir do arranjo L.

9.5 Exercícios

Reforço

R-9.1 Quantos prefixos não-vazios da cadeia P = `"aaabbaaa"` são também sufixos de P?

R-9.2 Desenhe uma figura ilustrando as comparações feitas pelo algoritmo de procura de padrões baseado em força bruta para o caso em que o texto é `"aaabaadaabaaa"` e o padrão é `"aabaaa"`.

R-9.3 Repita o problema anterior para o algoritmo BM de procura de padrões, não contando as comparações feitas para computar a função last(c).

R-9.4 Repita o problema anterior para o algoritmo KMP de procura de padrões, não contando as comparações feitas para computar a função de falha.

R-9.5 Compute uma tabela representando a função last usada no algoritmo BM para o padrão

`"the quick brown fox jumped over a lazy cat"`

assumindo o seguinte alfabeto (que começa com um espaço em branco):

$$\Sigma = \{\ ,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$$

R-9.6 Supondo que os caracteres no alfabeto Σ podem ser enumerados e indexar arranjos, forneça um método de tempo $O(m + |\Sigma|)$ para construir a função last a partir de um padrão P de comprimento m.

R-9.7 Compute uma tabela representando a função de falha KMP para o padrão `"cgtacgttcgtac"`.

R-9.8 Desenhe um *trie* padrão para a seguinte seqüência de cadeias de caracteres:

{abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca}.

R-9.09 Desenhe um *trie* comprimido para o conjunto de cadeias de caracteres do Exercício R-9.8.

R-9.10 Desenhe a representação compacta do *trie* de sufixos para a cadeia

`"minimize minime".`

R-9.11 Qual o mais longo prefixo da cadeia `"cgtacgttcgtacg"` que também é um sufixo desta cadeia?

R-9.12 Desenhe o arranjo das freqüências e a árvore de Huffman para a seguinte cadeia:

`"dogs do not spot hot pots or cats".`

R-9.13 Demonstre como usar programação dinâmica para computar a maior subseqüência comum entre as cadeias `"babbabab"` e `"bbabbaaab"`.

R-9.14 Demonstre o arranjo L da maior subseqüência comum para as duas cadeias

X = `"skullandbones"`
Y = `"lullabybabies"`

Qual é uma maior subseqüência comum entre essas cadeias de caracteres?

Criatividade

C-9.1 Dê um exemplo de um texto T de comprimento n e um padrão P de comprimento m que forem o algoritmo de procura de padrões por força bruta a um tempo de execução que seja $\Omega(mn)$.

C-9.2 Justifique por que o método KMPFailureFunction (Algoritmo 9.9) precisa de tempo $O(m)$ em um padrão de comprimento m.

C-9.3 Demonstre como modificar o algoritmo de procura de padrões KMP de forma que ele ache ***todas*** as ocorrências de um padrão P que apareça como *substring* em T, enquanto ainda estiver sendo executado em tempo $O(m + n)$. (Tenha certeza de encontrar até as ocorrências que se sobrepõem.)

C-9.4 Sejam T um texto de comprimento n e P um padrão de comprimento m. Descreva um método de tempo $O(m + n)$ para encontrar o prefixo mais longo de P que seja uma *substring* de T.

C-9.5 Dizemos que um padrão P de comprimento m é uma ***substring circular*** de um texto T de comprimento n se existir um índice $0 \le i < m$, tal que $P = T[n - m + i..n - 1] + T[0..i - 1]$, ou seja, se P for uma *substring* (normal) de T ou se P for igual à concatenação de um sufixo de T e um prefixo de T. Forneça um algoritmo de tempo $O(m + n)$ para determinar se P é uma *substring* circular de T.

C-9.6 O algoritmo de procura de padrões KMP pode ser modificado para maior velocidade em cadeias de caracteres binárias redefinindo a função de falha como

$$f(j) = \text{o maior } k < j \text{ tal que } P[0..k-2]\,\overline{P[k-1]} \text{ é sufixo de } P[1..j],$$

onde $\overline{P[k-1]}$ denota o complemento do k-ésimo *bit* de P. Descreva como implementar o algoritmo KMP para tirar vantagem dessa nova função de falha e forneça um método para avaliá-la. Demonstre que este método faz no máximo n comparações entre o texto e o padrão (contra $2n$ comparações do algoritmo KMP padrão, dado na Seção 9.1.4).

C-9.7 Modifique o algoritmo simplificado BM apresentado neste capítulo usando idéias do algoritmo KMP de forma que ele seja executado em tempo $O(m + n)$.

C-9.8 Demonstre como realizar procura de prefixos usando um *trie* de sufixos.

C-9.9 Forneça um algoritmo eficiente para deletar uma cadeia de caracteres de um *trie* padrão e analise seu tempo de execução.

C-9.10 Forneça um algoritmo eficiente para deletar uma cadeia de caracteres de um *trie* comprimido e analise seu tempo de execução.

C-9.11 Descreva um algoritmo para construir a representação compacta de um *trie* de sufixos e analise seu tempo de execução.

C-9.12 Seja T uma cadeia de caracteres de comprimento n. Descreva um método de tempo $O(n)$ para encontrar o mais longo prefixo de T que seja uma *substring* do reverso de T.

C-9.13 Descreva um algoritmo eficiente para encontrar o mais longo palíndromo que seja um sufixo de uma cadeia de caracteres T de comprimento n. Lembre que um palíndromo é uma cadeia de caracteres igual ao seu reverso. Qual o tempo de execução de seu método?

C-9.14 Dada uma seqüência $S = (x_0, x_1, x_2,..., x_{n-1})$ de números, descreva um algoritmo de tempo $O(n^2)$ para achar a mais longa subseqüência $T = (x_{i_0}, x_{i_1}, x_{i_2},..., x_{i_{k-1}})$ de números, onde $i_j < i_{j+1}$ e $x_{i_j} > x_{i_{j+1}}$. Ou seja, T é a mais longa subseqüência descendente de S.

C-9.15 Descreva um algoritmo de tempo $O(n \log(n))$ para o problema anterior.

C-9.16 Demonstre que uma seqüência de n números distintos contém uma subseqüência ascendente ou descendente de tamanho no mínimo igual a n.

C-9.17 Defina a ***distância de edição*** entre duas cadeias de caracteres X e Y de comprimentos n e m, respectivamente, como sendo o menor número de alterações para transformar X em Y. Uma alteração consiste em inserção de caracteres, deleção de caracteres ou substituição de caracteres. Por exemplo, as cadeias de caracteres `"algorithm"` e `"rhythm"` têm distância de edição 6. Projete um algoritmo de tempo $O(mn)$ para calcular a distância de edição entre X e Y.

C-9.18 Sejam A, B e C três cadeias de comprimento n com caracteres do alfabeto Σ. Projete um algoritmo de tempo $O(n^3)$ para encontrar a *substring* mais longa que seja comum a A, B e C.

C-9.19 Considere o algoritmo de procura de padrões que sempre diz "não" quando solicitado a determinar se um padrão P de tamanho m está contido em um texto T de tamanho n, ambos compostos de caracteres do mesmo alfabeto de d caracteres, com $d > 1$. Forneça uma caracterização em notação O da probabilidade de este algoritmo simples determinar incorretamente se P é ou não uma *substring* de T, assumindo que todas os padrões de comprimento m são igualmente prováveis. O limite deve ser $o(1)$.
Dica: este é um algoritmo bastante acurado à medida que m aumenta.

C-9.20 Sejam três arranjos de números inteiros A, B e C, cada um de comprimento n, representando os códigos ASCII ou Unicode para uma cadeia de caracteres. Dado um inteiro arbitrário x, apresente um algoritmo de tempo $O(n^2 \log(n))$ que determine se existem números $a \in A$, $b \in B$ e $c \in C$, tais que $x = a + b + c$.

C-9.21 Forneça um algoritmo de tempo $O(n^2)$ para o problema anterior.

C-9.22 Suponha que cada caracter c em um alfabeto $\sum$ tenha um valor inteiro $w(c)$. Dois jogadores irão para uma partida usando uma cadeia P de comprimento $2n$ sobre $\sum$. A cada rodada, o jogador deve selecionar e remover o primeiro (ou último) caractere em P, reduzindo seu comprimento em uma unidade. O objetivo de cada jogador é maximizar o prêmio total obtido com as somas dos caracteres que ele retirou. Forneça um algoritmo de tempo $O(n^2)$ para determinar a estratégia ótima de cada jogador.
Dica: use programação dinâmica.

C-9.23 Projete uma estratégia de empate ou vitória em tempo $O(n)$ para o primeiro jogador do exercício anterior. Sua estratégia não precisa ser ótima, mas deve garantir um empate ou vitória para o primeiro jogador.

Projetos

P-9.1 Realize uma análise experimental usando documentos retirados da Internet sobre a eficiência (número de comparações de caracteres realizadas) de pelo menos dois algoritmos de procura de padrões para padrões de diferentes comprimentos.

P-9.2 Implemente compressão e descompressão de documentos baseada na codificação de Huffman.

P-9.3 Crie uma classe que implemente um *trie* padrão para um conjunto de cadeias de caracteres ASCII. A classe deve ter um construtor que receba como argumento uma lista de cadeias de caracteres, e a classe deve ter um método que teste se uma dada cadeia de caracteres está armazenada no *trie*.

P-9.4 Implemente o mecanismo simplificado de busca descrito na Seção 9.2.4 para as páginas de um pequeno *website*. Use todas as palavras nas páginas do *site* como termos de índice, excluindo *stop words* como artigos, preposições e pronomes.

P-9.5 Implemente um mecanismo de busca para as páginas de um pequeno *website* adicionando um mecanismo de classificação de páginas ao mecanismo simplificado de busca descrito na Seção 9.2.4. A classificação de páginas deve retornar as páginas mais relevantes primeiramente. Use todas as palavras das páginas do *site* como termos de índice, excluindo *stop words* como artigos, preposições e pronomes.

P-9.6 Projete e implemente métodos que usem programação dinâmica e algoritmos gulosos para resolver o problema da maior subseqüência comum (LCS). Faça experimentos comparando os tempos de execução desses dois métodos com a qualidade da solução que eles produzem.

P-9.7 Implemente um algoritmo que receba qualquer cadeia de caracteres e produza uma codificação de Huffman para ela.

Notas

O algoritmo KMP é descrito por Knuth, Morris e Pratt em seu artigo [122]. Boyer e Moore descrevem seu algoritmo em um artigo publicado no mesmo ano [36]. Em seu artigo, no entanto, Knuth *et al*. [122] também provam que o algoritmo BM tem tempo linear. Mais recentemente, Cole [49] mostra que o algoritmo BM realiza no máximo $3n$ comparações no pior caso, e esse limite é exato. Todos os algoritmos discutidos acima são discutidos também no livro de Aho [6], embora com ênfase para a teoria. O leitor interessado em outros estudos sobre procura de padrões em cadeias de caracteres pode usar o livro de Stephen [193] e os capítulos de Aho [6] e Crochemore e Lecroq [56].

O *trie* foi inventado por Morrison [156] e é amplamente discutido no livro clássico *Sorting and Searching*, de Knuth [119]. O nome "Patricia" abrevia "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [156]. McCreight [139] mostra como construir *tries* de sufixos em tempo linear. Uma introdução ao campo da recuperação de informação que inclui uma discussão de mecanismos de busca para a Web é dada no livro de Baeza-Yates e Ribeiro-Neto [21]. A aplicação do método guloso para o problema da codificação é originária de Huffman [104].

Capítulo

Teoria dos números e criptografia 10

10.1 Algoritmos fundamentais envolvendo números 449

10.1.1 Alguns fatos da teoria elementar dos números 449

10.1.2 O algoritmo de GCD de Euclides 451

10.1.3 Aritmética modular 453

10.1.4 Exponenciação modular 457

10.1.5 Inversos multiplicativos modulares 459

10.1.6 Testes de primalidade 460

10.2 Processos criptográficos 465

10.2.1 Esquemas simétricos de encriptação 467

10.2.2 Esquemas de chave pública 469

10.2.3 O esquema RSA 470

10.2.4 O esquema El Gamal 473

10.3 Algoritmos e protocolos para segurança de informações 474

10.3.1 Funções *hash* unidirecionais 475

10.3.2 Carimbos de tempo e dicionários autenticados 475

10.3.3 Lançamento de moedas e compromisso de *bits* 476

10.3.4 Protocolo para transações eletrônicas seguras (SET) 477

10.3.5 Distribuição de chaves e seu intercâmbio 479

10.4 A transformada rápida de Fourier 481

10.4.1 Raízes primitivas da unidade 483

10.4.2 A transformada discreta de Fourier 484

10.4.3 O algoritmo da transformada rápida de Fourier 487

10.4.4 Multiplicando inteiros grandes 490

10.5 Exemplo em Java: a FFT 492

10.6 Exercícios 500

Os computadores são usados hoje em uma variedade de aplicações que exigem sigilo. Clientes usam comércio eletrônico para fazer compras e pagar suas contas. Empresas usam a Internet para compartilhar documentos sigilosos e iteragir com parceiros. Universidades usam redes de computadores para armazenar informações pessoais sobre os estudantes e suas notas. Esse tipo de informação sigilosa pode ser potencialmente danosa se for alterada, destruída ou cair em mãos erradas. Neste capítulo, discutiremos várias técnicas algorítmicas poderosas para proteger informações sigilosas e atingir as seguintes metas:

- ***Integridade de dados:*** a informação não deve ser alterada sem que isso seja detectado. Por exemplo, é importante evitar a modificação de uma ordem de compra ou outros documentos contratuais que sejam transmitidos eletronicamente.
- ***Autenticação:*** indivíduos e organizações que estão acessando ou comunicando informação sigilosa devem ser corretamente identificados, ou seja, autenticados. Por exemplo, corporações que oferecem facilidades de rede a seus empregados devem prover um procedimento de autenticação para que eles acessem bases de dados pela Internet.
- ***Autorização:*** agentes que estão realizando computações que envolvem informações sigilosas devem ser autorizados a realizar essas computações.
- ***Irrecusabilidade:*** em transações que implicam um contrato, as partes que aderiram ao contrato não devem poder negar suas obrigações sem ser detectadas.
- ***Confidencialidade:*** informações sigilosas devem ser mantidas secretas para indivíduos que não estão autorizados a vê-las. Ou seja, devemos garantir que os dados são vistos pelo remetente e pelo destinatário, mas não por partes não-autorizadas que possam espionar a comunicação. Por exemplo, muitas mensagens de correio eletrônico devem ser mantidas confidenciais.

Muitas das técnicas que discutiremos neste capítulo para atingir as metas acima utilizam a teoria dos números. Assim, começamos este capítulo discutindo uma série de conceitos e algoritmos importantes na teoria dos números. Descreveremos o milenar e surpreendentemente eficiente algoritmo de Euclides para determinar o maior divisor comum entre dois números, bem como algoritmos para determinar expoentes modulares e inversas. Além disso, como os números primos têm um papel central em aplicações criptográficas, discutiremos métodos eficientes para testar se um número é primo. Mostraremos de que forma esses algoritmos da teoria dos números podem ser usados em algoritmos criptográficos que implementam facilidades de segurança, concentrando-nos em criptografia e assinaturas digitais, o que inclui o popular esquema RSA. Também discutiremos neste capítulo vários protocolos que podem ser construídos com esses algoritmos.

Concluiremos discutindo a transformada rápida de Fourier (TRF, ou FFT* em inglês), uma técnica de divisão e conquista que pode resolver muitos problemas com propriedades similares à multiplicação. Mostraremos como usar a FFT para multiplicar eficientemente polinômios e inteiros longos. Também forneceremos uma implementação em Java do algoritmo da FFT para multiplicar inteiros longos e comparamos empiricamente o desempenho deste algoritmo com a técnica comum de multiplicação de inteiros.

* N. de R. *Fast Fourier Transform*.

10.1 Algoritmos fundamentais envolvendo números

Nesta seção, discutimos vários algoritmos fundamentais para realizar operações importantes envolvendo números. Descrevemos métodos eficientes para determinar expoentes módulo n, para calcular inversas multiplicativas módulo n e para testar se um inteiro n é primo. Todas essas computações têm várias aplicações importantes, incluindo a formação dos algoritmos críticos em várias aplicações criptográficas. Antes que possamos apresentar esses algoritmos, devemos primeiramente apresentar alguns fatos básicos da teoria dos números. Nesta discussão, assumiremos que todas as variáveis são inteiras, e as provas de alguns fatos matemáticos serão deixadas como exercícios.

10.1.1 Alguns fatos da teoria elementar dos números

Para começar, precisamos de alguns fatos da teoria elementar dos números, incluindo alguma notação e definições. Dados inteiros positivos a e b, usamos a notação

$$a|b$$

para indicar que a ***divide*** b, ou seja, b é um multiplo de a. Se $a|b$, então sabemos que existe um inteiro k tal que $b = ak$. As propriedades a seguir da divisibilidade se seguem imediatamente da definição:

Teorema 10.1: *sejam a, b e c inteiros arbitrários. Então*

- *Se $a|b$ e $b|c$, então $a|c$.*
- *Se $a|b$ e $a|c$, então $a|(ib + jc)$ para todos os inteiros i e j.*
- *Se $a|b$ e $b|a$, então $a = b$ ou $a = -b$.*

Prova: veja o Exercício R-10.1.

Um inteiro p é ***primo*** se $p \geq 2$ e se seus únicos divisores são os divisores triviais 1 e p. Assim, se p é primo, então $d|p$ implica que $d = 1$ ou $d = p$. Um inteiro maior do que 2 que não é primo é dito ***composto***. Assim, por exemplo, 5, 11, 101 e 98.711 são primos, enquanto 25 e 10.403 (=101·103) são compostos. Também temos o seguinte:

Teorema 10.2 (Teorema Fundamental da Aritmética): *seja $n > 1$ um inteiro. Então existe um único conjunto de números primos $\{p_1,\ldots, p_k\}$ e expoentes inteiros positivos $\{e_1,\ldots, e_k\}$ tais que*

$$n = p_1^{e_1} \cdots p_k^{e_k}.$$

O produto $n = p_1^{e_1} \cdots p_k^{e_k}$ é conhecido como a ***decomposição em fatores primos*** de n, neste caso. O Teorema 10.2 e a noção de unicidade da decomposição em fatores primos são a base de vários algoritmos criptográficos.

O Maior Divisor Comum (MDC)

O ***maior divisor comum*** de dois inteiros positivos a e b, denotado por $\gcd(a, b)$, é o maior inteiro que divide a e b. Alternativamente podemos dizer que $\gcd(a, b)$ é o número c tal que, se $d|a$ e $d|b$, então $d|c$. Se $\gcd(a, b) = 1$, então dizemos que a e b são ***re-***

* N. de T. Em inglês, GCD – *greatest common divisor*.

lativamente primos. Estendemos a noção de maior divisor comum para um par de inteiros arbitrários pelas seguintes regras:

- $\gcd(a, 0) = \gcd(0, a) = a$.
- $\gcd(a, b) = \gcd(|a|, |b|)$, o que cuida dos valores negativos.

O operador de módulo

Algumas palavras sobre o ***operador de módulo*** (mod) são apropriadas. Lembre que $a \bmod n$ é o resto da divisão de a por n. Ou seja,

$$r = a \bmod n$$

significa que

$$r = a - \lfloor a/n \rfloor n.$$

Em outras palavras, existe um inteiro q tal que

$$a = qn + r.$$

Note, além disso, que $a \bmod n$ sempre é um inteiro no conjunto $\{0, 1, 2,..., n-1\}$, mesmo quando a é negativo.

Às vezes é conveniente falar sobre a ***congruência*** módulo n. Se

$$a \bmod n = b \bmod n,$$

dizemos que a é ***congruente*** a b módulo n, que chamamos de ***modulus***, e escrevemos

$$a \equiv b \pmod{n}$$

Portanto, se $a \equiv b \bmod n$, então $a - b = kn$ para algum inteiro k.

Relacionando o operador de módulo e o GCD

O teorema a seguir fornece uma caracterização alternativa do maior divisor comum. Sua prova faz uso do operador de módulo.

Teorema 10.3: *para quaisquer inteiros positivos a e b, o $\gcd(a, b)$ é o menor inteiro positivo d tal que $d = ia + bj$ para inteiros i e j. Em outras palavras, se d for a menor combinação linear inteira positiva de a e b, então $d = \gcd(a, b)$.*

Prova: suponha que d seja o menor inteiro tal que $d = ia + bj$ para inteiros i e j. Note que, pela definição de d, qualquer divisor comum de a e b também é um divisor de d. Assim, $d \geq \gcd(a, b)$. Para completar a prova, precisamos mostrar que $d \leq \gcd(a, b)$.

Seja $h = \lfloor a/d \rfloor$. Ou seja, h é o inteiro tal que $a \bmod d = a - hd$. Então,

$$\begin{aligned} a \bmod d &= a - hd \\ &= a - h(ia + jb) \\ &= (1 - hi)a + (-hj)b. \end{aligned}$$

Em outras palavras, $a \bmod d$ também é uma combinação linear inteira de a e b. Além disso, pela definição do operador de módulo, $a \bmod d < d$. Mas d é a menor combinação linear inteira positiva de a e b. Assim, devemos concluir que $a \bmod d = 0$, o que implica $d|a$. Adicionalmente, por um argumento similar, temos que $d|b$. Assim, d é um divisor de a e b, o que implica $d \leq \gcd(a, b)$. ■

Como mostraremos na Seção 10.1.3, esse teorema mostra que a função gcd é útil para determinar inversas multiplicativas modulares. Na próxima subseção, mostraremos como calcular rapidamente a função gcd.

10.1.2 O algoritmo de GCD de Euclides

Para determinar o máximo divisor comum de dois números, podemos usar um dos mais antigos algoritmos conhecidos, o algoritmo de Euclides. Esse algoritmo é baseado na seguinte propriedade do gcd(a, b):

Lema 10.4: *sejam a e b inteiros positivos. Para qualquer inteiro r, temos*

$$\gcd(a, b) = \gcd(b, a - rb)$$

Prova: seja $d = \gcd(a, b)$ e $c = \gcd(b, a - rb)$. Ou seja, d é o maior inteiro tal que $d|a$ e $d|b$, e c é o maior inteiro tal que $c|b$ e $c|(a - rb)$. Queremos provar que $d = c$. Pela definição de d, o número

$$(a - rb)/d = a/d - r\,(b/d).$$

é um inteiro. Assim, d divide a e $a - rb$; portanto, $d \leq c$.

Pela definição de c, $k = b/c$ é um inteiro, pois $c|b$. Além disso, o número

$$(a - rb)/c = a/c - rk$$

também deve ser um inteiro, pois $c|(a - rb)$. Assim a/c também deve ser um inteiro, ou seja, $c|a$. Portanto, c divide a e b, e por isso $c \leq d$. Concluímos então que $d = c$. ■

O Lema 10.4 leva facilmente a um antigo algoritmo, conhecido como o algoritmo de Euclides, para determinar o máximo divisor comum (GCD) de dois números, mostrado no Algoritmo 10.1.

Algoritmo EuclidGCD(a, b):
  Entrada: inteiros não-negativos a e b.
  Saída: gcd(a, b).
  se $b = 0$ **então**
    retorne a
  retorne EuclidGCD(b, a mod b)

Algoritmo 10.1 Algoritmo de Euclides para o GCD.

Um exemplo da execução do algoritmo de Euclides é mostrado na Tabela 10.2.

	1	2	3	4	5	6	7
a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0

Tabela 10.2 Um exemplo da execução do algoritmo de Euclides para determinar gcd(412, 260) = 4. Os argumentos a e b de cada chamada recursiva do método EuclidGCD(412,260) são mostradas da direita para a esquerda, com as colunas mostrando o nível de recursão do método EuclidGCD.

Análise do algoritmo de Euclides

O número de operações aritméticas usadas pelo método EuclidGCD(a, b) é proporcional ao número de chamadas recursivas. Portanto, para limitar o número de operações aritméticas usadas pelo algoritmo de Euclides, precisamos apenas limitar o número de chamadas recursivas. Primeiramente, observamos que, após a primeira chamada, o primeiro argumento é sempre maior do que o segundo. Para $i > 0$, seja a_i o primeiro argumento da i -ésima chamada recursiva do método EuclidGCD. É claro que o segundo argumento da chamada é igual a a_{i+1}, o primeiro argumento da próxima chamada. Também temos

$$a_{i+2} = a_i \bmod a_{i+1},$$

implicando que a seqüência dos a_i é estritamente decrescente. Agora mostraremos que a seqüência decresce rapidamente. Especificamente, afirmamos que

$$a_{i+2} \leq \frac{1}{2} a_i.$$

Para provar a afirmação, diferenciamos dois casos:

Caso 1: $a_{i+2} \leq \frac{1}{2} a_i$. Como a seqüência dos a_i é estritamente decrescente, temos

$$a_{i+2} < a_{i+1} \leq \frac{1}{2} a_i.$$

Caso 2: $a_{i+2} > \frac{1}{2} a_i$. Neste caso, já que $a_{i+2} = a_i \bmod a_{i+1}$, temos

$$a_{i+2} = a_i \mod a_{i+1} = a_i - a_{i+1} < \frac{1}{2} a_i.$$

Desta forma, o primeiro argumento do método EuclidGCD decresce a metade de seu valor a cada chamada recursiva. Portanto, podemos resumir a análise acima da forma a seguir.

Teorema 10.5: *sejam a e b inteiros positivos. O algoritmo de Euclides calcula o* gcd(a, b) *executando* $O(\log \max(a, b))$ *operações aritméticas.*

Salientamos que o limite de complexidade foi obtido contando-se as operações aritméticas. Podemos melhorar o fator constante no limite acima tirando vantagem do fato de que, atualmente, o algoritmo de Euclides deve ser implementado em um computador digital.

O algoritmo binário de Euclides

Uma variação do algoritmo de Euclides, chamada de ***algoritmo binário de Euclides***, leva em conta o fato de que a divisão por 2 em um computador digital é mais rápida do que a divisão por um inteiro qualquer, pois ela pode ser realizada por uma única instrução de ***shift-à-direita***. O algoritmo binário de Euclides é mostrado no Algoritmo 10.3. Como o algoritmo de Euclides original, ele determina o máximo divisor comum de dois inteiros a e b com $O(\log \max(a, b))$ operações aritméticas, mas com um fator

constante menor. A justificativa de seu funcionamento e a análise assintótica de seu tempo de execução são exploradas no Exercício C-10.1.

Algoritmo EuclidBinaryGCD(a, b):
  Entrada: inteiros não-negativos a e b.
  Saída: gcd(a, b).
  se $a = 0$ **então**
    retorne b
  senão se $b = 0$ **então**
    retorne a
  senão se a é par e b é par **então**
    retorne 2. EuclidBinaryGCD($a/2$, $b/2$)
  senão se a é par e b é ímpar **então**
    retorne EuclidBinaryGCD($a/2$, b)
  senão se a é ímpar e b é par **então**
    retorne EuclidBinaryGCD(a, $b/2$)
  senão
    {a é ímpar e b é ímpar }
    retorne EuclidBinaryGCD($|a - b|/2$, min(a/b)

Algoritmo 10.3 Algoritmo binário de Euclides para determinar o maior divisor comum de dois inteiros não-negativos.

10.1.3 Aritmética modular

Seja Z_n o conjunto de inteiros não-negativos menores do que n:

$$Z_n = \{0, 1, ..., (n - 1)\}.$$

O conjunto Z_n também é chamado de conjunto dos ***resíduos*** módulo n, porque, se $b = a \bmod n$, então b é às vezes chamado de ***resíduo*** de a módulo n. A aritmética modular em Z_n, onde as operações são realizadas sobre os elementos de Z_n e os resultados são obtidos módulo n, tem propriedades semelhantes às da aritmética tradicional, como associatividade, comutatividade, distributividade da adição e da multiplicação e a existência de elementos neutros 0 e 1 para adição e multiplicação, respectivamente. Além disso, em qualquer expressão aritmética, reduzir as subexpressões módulo n produz o mesmo resultado que calcular a expressão completa e depois reduzir o valor módulo n. Ademais, todo elemento x em Z_n tem um ***inverso aditivo***, ou seja, para cada $x \in Z_n$ existe um $y \in Z_n$ tal que $x + y \bmod n = 0$. Por exemplo, o inverso aditivo de 5 módulo 11 é 6.

Quando se trata de inversos multiplicativos, no entanto, surge uma importante diferença. Seja x um elemento de Z_n. Um ***inverso multiplicativo*** de x é um elemento $x^{-1} \in Z_n$ tal que $xx^{-1} \equiv 1 \bmod n$. Por exemplo, o inverso multiplicativo de 5 módulo 9 é 2, ou seja, $5^{-1} = 2$ em Z_9. Como na aritmética habitual, 0 não tem um inverso multiplicativo em Z_n. Mesmo assim, alguns elementos diferentes de zero podem também não ter inversos multiplicativos em Z_n. Por exemplo, 3 não tem um inverso multiplicativo em Z_9. Entretanto, se n é primo, então todo elemento $x \neq 0$ em Z_n tem um inverso multiplicativo em Z_n (1 é seu próprio inverso multiplicativo).

Teorema 10.6: *um elemento $x > 0$ de Z_n tem um inverso multiplicativo em Z_n se e somente se* $\gcd(x, n) = 1$ *(ou seja, se $x = 1$ ou se x não divide n).*

Prova: suponha que $\gcd(x, n) = 1$. Pelo Teorema 10.3, existem inteiros i e j tais que $ix + jn = 1$. Isso implica $ix \bmod n = 1$, ou seja, $i \bmod n$ é o inverso multiplicativo de x em Z_n, o que prova a parte do "se" do teorema.

Para provar a parte do "somente se", suponha, para obtermos uma contradição, que $x > 1$ divide n, e existe um elemento y tal que $xy \equiv 1 \bmod n$. Temos $xy = kn + 1$ para algum inteiro k. Assim, encontramos inteiros $i = y$ e $j = -k$ tais que $ix + jn = 1$. Pelo Teorema 10.3, isso implica $\gcd(x, n) = 1$, o que é uma contradição. ■

Se $\gcd(x, n) = 1$, dizemos que x e n são ***relativamente primos*** (1 é relativamente primo a todos os outros números). Assim, o Teorema 10.6 implica que x tem um inverso multiplicativo em Z_n se e somente se x é relativamente primo a n. Além disso, o Teorema 10.6 implica que a seqüência $0, x, 2x, 3x,..., (n-1)x$ é simplesmente uma reordenação dos elementos de Z_n, ou seja, uma permutação dos elementos de Z_n, como mostrado a seguir.

Corolário 10.7: *seja $x > 0$ um elemento de Z_n tal que* $\gcd(x, n) = 1$. *Então*

$$Z_n = \{ix : i = 0, 1, ..., n-1\}$$

Prova: veja o Exercício R-10.7. ■

Na Tabela 10.4, mostramos os inversos multiplicativos dos elementos de Z_{11} como um exemplo. Quando o inverso multiplicativo x^{-1} de x existe em Z_n, a notação y/x em uma expressão tomada módulo n significa "$yx^{-1} \bmod n$".

x	0	1	2	3	4	5	6	7	8	9	10
$x^{-1} \bmod 11$		1	6	4	3	9	2	8	7	5	10

Tabela 10.4 Inversos multiplicativos dos elementos de Z_{11}.

O pequeno teorema de Fermat

Neste momento, temos equipamento suficiente para nosso primeiro teorema de maior relevância, conhecido como o ***pequeno teorema de Fermat***.

Teorema 10.8 (pequeno teorema de Fermat): *seja p um número primo e seja x um inteiro tal que $x \bmod p \neq 0$. Então*

$$x^{p-1} \equiv 1 \pmod{p}.$$

Prova: é suficiente provar o resultado para $0 < x < p$, pois

$$x^{p-1} \bmod p = (x \bmod p)^{p-1} \bmod p,$$

pois podemos reduzir cada subexpressão "x" em "x^{p-1}" módulo p.

Pelo Corolário 10.7, sabemos que, para $0 < x < p$, o conjunto $\{1, 2,..., p-1\}$ e o conjunto $\{x \cdot 1, x \cdot 2,..., x \cdot (p-1)\}$ contêm exatamente os mesmos elementos. Assim, quando multiplicamos os elementos dos conjuntos uns pelos outros, obtemos o mesmo valor, que é

$$1 \cdot 2 \cdots (p-1) = (p-1)!.$$

Em outras palavras,

$$(x \cdot 1) \cdot (x \cdot 2) \cdots (x \cdot (p-1)) \equiv (p-1)! \pmod{p}.$$

Se fatorarmos os termos x, temos

$$x^{p-1}(p-1)! \equiv (p-1)! \pmod{p}.$$

Como p é primo, cada elemento não-nulo de Z_p tem um inverso multiplicativo. Assim, podemos cancelar o termo $(p-1)!$ de ambos os lados, o que resulta em $x^{p-1} \equiv 1 \bmod p$, o resultado desejado. ■

Na Tabela 10.5, mostramos as potências dos elementos não-nulos de Z_{11}. Observe os seguintes padrões interessantes:

- A última coluna da tabela, com os valores de $x^{10} \bmod 11$ para $x = 1,\cdots, 10$, contém apenas valores 1, como assegurado pelo pequeno teorema de Fermat.
- Na linha 1, uma subseqüência de um elemento (1) é repetida 10 vezes.
- Na linha 10, uma subseqüência de dois elementos terminando em 1 é repetida cinco vezes, pois $10^2 \bmod 11 = 1$.
- Nas linhas 3, 4, 5 e 9, uma subseqüência de cinco elementos terminando em 1 é repetida duas vezes.
- Em cada uma das linhas 2, 6, 7 e 8, os 10 elementos são diferentes.
- Os comprimentos das subseqüências formando as linhas da tabela e seus números de repetições são os divisores de 10, ou seja, 1, 2, 5 e 10.

x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}
1	1	1	1	1	1	1	1	1	1
2	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6	3	7	9	10	5	8	4	2	1
7	5	2	3	10	4	6	9	8	1
8	9	6	4	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

Tabela 10.5 Potências sucessivas dos elementos de Z_{11} módulo 11.

O teorema de Euler

A função ***totiente*** de Euler de um inteiro positivo n, denotada por $\phi(n)$, é definida como o número de inteiros positivos menores ou iguais a n e que são relativamente primos a n. Ou seja, $\phi(n)$ é igual ao número de elementos em Z_n que têm inversos multiplicativos em Z_n. Se p é primo, então $\phi(p) = p - 1$. De fato, como p é primo, cada um dos números $1, 2,..., p-1$ é relativamente primo a p, e $\phi(p) = p - 1$.

O que acontece se n não for um número primo? Suponha que $n = pq$, onde p e q são primos. Quantos números são relativamente primos a n? Inicialmente observamos que existem pq inteiros entre 1 e n. Entretanto, q deles (incluindo n) são múltiplos de p, e então existe um máximo divisor comum diferente de 1 entre p e n. De forma similar, existem p múltiplos de q (de novo, incluindo n). Esses múltiplos não podem ser contados em $\phi(n)$; portanto, percebemos que

$$\phi(n) = pq - q - (p - 1) = (p - 1)(q - 1).$$

A função totiente de Euler é relacionada a um importante subconjunto de Z_n, conhecido como o ***grupo multiplicativo*** de Z_n, que é denotado como Z_n^*. O conjunto Z_n^* é definido como o conjunto de inteiros entre 1 e n que são relativamente primos a n. Se n é primo, então Z_n^* consiste nos $n - 1$ elementos não-nulos em Z_n, ou seja, $Z_n^* = \{1, 2,..., n - 1\}$ se n é primo. Em geral, Z_n^* contém $\phi(n)$ elementos.

O conjunto Z_n^* possui várias propriedades interessantes, e uma das mais importantes é o fato de que ele é fechado para a multiplicação módulo n. Ou seja, para qualquer par de elementos a e b em Z_n^*, temos que $c = ab \bmod n$ também está em Z_n^*. De fato, pelo Teorema 10.6, a e b têm inversos multiplicativos em Z_n^*. Para perceber que Z_n^* tem essa propriedade, seja $d = a^{-1}b^{-1} \bmod n$. Então, $cd \bmod n = 1$, implicando que d é o inverso multiplicativo de c em Z_n. Assim, aplicando de novo o Teorema 10.6, temos que c é relativamente primo a n, ou seja, $c \in Z_n^*$. Em termos algébricos, dizemos que Z_n^* é um ***grupo***, uma forma de dizer que cada elemento em Z_n^* tem um inverso multiplicativo, e a multiplicação em Z_n^* é associativa, tem uma identidade e é fechada em Z_n^*.

O fato de que Z_n^* contém $\phi(n)$ elementos e é um grupo multiplicativo leva naturalmente a uma extensão do pequeno teorema de Fermat. Lembre-se de que, no pequeno teorema de Fermat, o expoente é $p - 1 = \phi(p)$, já que p é primo. Em seguida, pode-se generalizar o teorema, e esta forma generalizada é apresentada no teorema a seguir, conhecido como o ***teorema de Euler***.

Teorema 10.9 (teorema de Euler): *seja n um inteiro positivo e x um inteiro tal que* $\gcd(x, n) = 1$. *Então*

$$x^{\phi(n)} \equiv 1 \qquad (\bmod\ n)$$

Prova: a técnica de prova é similar à do pequeno teorema de Fermat. Denote os elementos do conjunto Z_n^*, o grupo multiplicativo de Z_n, como $u_1, u_2,..., u_{\phi(n)}$. Pela propriedade do fechamento de Z_n^* e pelo Corolário 10.7,

$$Z_n^* = \{xu_i : i = 1, \cdots, \phi(n)\},$$

ou seja, multiplicar os elementos de Z_n^* por x módulo n simplesmente permuta a seqüência $u_1, u_2,..., u_{\phi(n)}$. Assim, multiplicando todos os elementos de Z_n^*, obtemos

$$(xu_1) \cdot (xu_2) \cdots (xu_{\phi(n)}) \equiv u_1 u_2 \cdots u_{\phi(n)} \qquad (\bmod\ n)$$

Novamente, agrupamos o termo $x^{\phi(n)}$ de um lado e temos a congruência

$$x(u_1 u_2 \cdots u_{\phi(n)}) \equiv u_1 u_2 \cdots u_{\phi(n)} \qquad (\bmod\ n).$$

Dividindo pelo produto dos u_i temos $x^{\phi(n)} \equiv 1 \bmod n$. ■

O Teorema 10.9 fornece uma expressão fechada para os inversos multiplicativos, ou seja, se x e n são relativamente primos, podemos escrever

$$x^{-1} \equiv x^{\phi(n)-1} \pmod{n}.$$

Geradores

Dado um primo p e um inteiro a entre 1 e $p - 1$, a ***ordem*** de a é o menor expoente $e > 1$ tal que

$$a^e \equiv 1 \bmod q.$$

Um ***gerador*** (também chamado de ***raiz primitiva***) de Z_p é um elemento g de Z_p com ordem $p - 1$. Usamos o termo "gerador" para um tal elemento a porque a exponenciação repetida de a pode gerar todo o conjunto Z_p^*. Por exemplo, como mostrado na Tabela 10.5, os geradores de Z_{11} são 2, 6, 7 e 8. Os geradores têm um papel importante em muitas computações, incluindo o algoritmo da Transformada Rápida de Fourier discutido na Seção 10.4. A existência de geradores é estabelecida pelo teorema a seguir, enunciado sem sua prova.

Teorema 10.10: *se p é um número primo, então o conjunto Z_p tem $\phi(p - 1)$ geradores.*

10.1.4 Exponenciação modular

Começaremos tratando do problema da exponenciação. A questão principal neste caso é achar um método diferente do de força bruta. Antes de descrever um algoritmo eficiente, no entanto, vamos rever o algoritmo mais óbvio, pois ele contém uma técnica importante.

Exponenciação por força bruta

Uma das considerações mais importantes em qualquer algoritmo de exponenciação é evitar que os resultados intermediários sejam muito grandes. Suponha que desejamos calcular $30.192^{43791} \bmod 65.301$. Multiplicar 30.192 por si mesmo 43.791 vezes e ***depois*** realizar a operação de módulo fornecerá resultados imprevisíveis na maioria das linguagens de programação devido aos números imensos que estão envolvidos. Assim, devemos aplicar o módulo a cada iteração, como mostrado no Algoritmo 10.6.

Algoritmo NaiveExponentiation(a, p, n):
  Entrada: inteiros a, p e n.
  Saída: $r = a^p \bmod n$.
$r \leftarrow 1$
para $i \leftarrow 1$ **a** p **faça**
  $r \leftarrow (r \cdot a) \bmod n$
retorna r

Algoritmo 10.6 Um método de força bruta para exponenciação modular.

Esse algoritmo "óbvio" de exponenciação modular está correto mas não é muito eficiente, pois custa $\Theta(p)$ iterações para determinar a exponenciação modular de um número na potência p. Com expoentes grandes, o tempo de execução é bastante alto. Felizmente, existe um método melhor.

O método dos quadrados sucessivos

Uma observação simples mas importante para um algoritmo aprimorado de exponenciação modular é a de que elevar um número a^p ao quadrado equivale a multiplicar seu expoente por dois. Além disso, multiplicar dois números a^p e a^q equivale a determinar $a^{(p+q)}$. Vamos portanto escrever o expoente p como um número binário $p_{b-1} \cdots p_0$, ou seja,

$$p = p_{b-1}2^{b-1} + \cdots p_0 2^0.$$

É claro que cada um dos p_i é 0 ou 1. Usando a observação acima, podemos determinar $a^p \bmod n$ com uma variação da regra de Horner usada para avaliar polinômios, onde o polinômio em questão é a expansão binária do expoente p. Especificamente, definimos q_i como o número cuja representação binária é dada pelos i *bits* mais à esquerda de p, ou seja, q_i é escrito em binário como $p_{b-1}...p_{b-i}$. Claramente, temos $p = q_b$. Note que $q_1 = p_{b-1}$, e podemos definir q_i recursivamente como

$$q_i = 2q^{i-1} + p_{b-i} \quad \text{para} \quad 1 < i \leq b.$$

Assim podemos avaliar $a^p \bmod n$ com uma computação recursiva, chamada de método dos ***quadrados sucessivos***, mostrado no Algoritmo 10.7.

A idéia principal deste algoritmo é considerar um *bit* do expoente p de cada vez, dividindo p por dois até que p seja zero, o que eleva ao quadrado o produto Q_i para cada *bit*. Além disso, se o *bit* é 1 (ou seja, se p é ímpar), então multiplicamos a base a também. Para perceber por que este algoritmo funciona, definimos, para $i = 1,..., b$,

$$Q_i = a^{q_i} \bmod n.$$

Pela definição recursiva de q_i, derivamos a seguinte definição de Q_i:

$$Q_i = (Q_{i-1}^2 \bmod n)a^{p_{b-i}} \bmod n \quad \text{para} \quad 1 < i \leq b. \tag{10.1}$$

$$Q_1 = a^{p_{b-1}} \bmod n.$$

É fácil verificar que $Q_b = a^p \bmod n$.

Algoritmo FastExponentiation(a, p, n):
  Entrada: inteiros a, p e n.
  Saída: $r = a^p \bmod n$.
 se $p = 0$ **então**
  retorne 1
 se p é par **então**
  $t \leftarrow$ FastExponentiation(a, $p/2$, n) { p é par, então $t = a^{p/2} \bmod n$ }
  retorne $t^2 \bmod n$
 $t \leftarrow$ FastExponentiation(a, $(p-1)/2$, n) { p é ímpar, então $t = a^{(p-1)/2} \bmod n$ }
 retorne $a(t^2 \bmod n) \bmod n$

Algoritmo 10.7 O algoritmo FastExponentiation para exponenciação modular usando o método dos quadrados sucessivos. Note que, como o operador de módulo é aplicado a cada operação aritmética no método FastExponentiation, o tamanho dos operandos de cada operação de módulo e de multiplicação nunca é maior do que $2\lceil \log_2(n) \rceil$ *bits*.

p	12	6	3	1	0
r	1	12	8	2	1

Tabela 10.8 Exemplo da execução do algoritmo de quadrados sucessivos para exponenciação modular. Para cada chamada recursiva de FastExponentiation(2,12,13), mostramos o segundo argumento p e o valor do resultado $r = 2^p \bmod 13$.

Mostramos uma execução do método dos quadrados sucessivos para exponenciação modular na Tabela 10.8.

O tempo de execução do algoritmo dos quadrados sucessivos é fácil de analisar. No Algoritmo 10.7, um número constante de operações aritméticas é realizado, excluindo aquelas da chamada recursiva. Em cada chamada recursiva, o expoente p é dividido por dois. Assim, o número de chamadas recursivas e operações aritméticas é $O(\log(p))$. Podemos, portanto, resumir como segue.

Teorema 10.11: *sejam a, p e n inteiros positivos com $a < n$. O algoritmo dos quadrados sucessivos determina $a^p \bmod n$ usando $O(\log(p))$ operações aritméticas.*

10.1.5 Inversos multiplicativos modulares

Vamos abordar agora o problema de computar inversos multiplicativos em Z_n. Primeiramente recordemos o Teorema 10.6, que diz que um elemento não-negativo x de Z_n tem um inverso se e somente se $\gcd(x, n) = 1$. A prova do Teorema 10.6 sugere uma forma de determinar $x^{-1} \bmod n$ se acharmos os inteiros i e j usados pelo Teorema 10.3 tais que

$$ix + jn = \gcd(x, n) = 1.$$

Se encontrarmos os inteiros i e j, imediatamente obtemos

$$i \equiv x^{-1} \bmod n.$$

A determinação dos inteiros i e j usados pelo Teorema 10.3 pode ser feita com uma variação do algoritmo de Euclides, chamada de ***algoritmo estendido de Euclides***.

O algoritmo estendido de Euclides

Sejam a e b inteiros positivos, e denote com d seu maior divisor comum:

$$d = \gcd(a, b).$$

Seja $q = a \bmod b$ e seja r o inteiro tal que $a = rb + q$, ou seja,

$$q = a - rb.$$

O algoritmo de Euclides é baseado na aplicação repetida da fórmula

$$d = \gcd(a, b) = \gcd(b, q),$$

que resulta imediatamente no Lema 10.4.

Suponha que a chamada recursiva com argumentos b e q também retorna inteiros k e l tais que

$$d = kb + lq.$$

Lembrando da definição de r, temos

$$d = kb + lq = kb + l(a - rb) = la + (k - lr)b.$$

Assim temos

$$d = ia + jb, \text{ para } i = l \text{ e } j = k - lr.$$

Esta última equação sugere um método de determinar os inteiros i e j. Este método, conhecido como algoritmo estendido de Euclides, é mostrado no Algoritmo 10.9. Apresentamos na Tabela 10.10 uma execução desse algoritmo. Sua análise é análoga à do algoritmo de Euclides.

Teorema 10.12: *sejam a e b inteiros positivos. O algoritmo estendido de Euclides para determinar uma tripla de inteiros (d, i, j) tais que*

$$d = \gcd(a, b) = ia + jb,$$

executa $O(\log \max(a, b))$ operações aritméticas.

Corolário 10.13: *seja x um elemento de Z_n tal que* $\gcd(x, n) = 1$. *O inverso multiplicativo de x em Z_n pode ser determinado com $O(\log(n))$ operações aritméticas.*

Algoritmo ExtendedEuclidGCD(a, b):
  Entrada: inteiros não-negativos a e b.
  Saída: Tripla de inteiros (d, i, j) tais que $d = \gcd(a, b) = ia + jb$.
se $b = 0$ **então**
  retorna $(a, 1, 0)$;
$q \leftarrow a \bmod b$
Seja r o inteiro tal que $a = rb + q$
$(d, k, l) \leftarrow$ ExtendedEuclidGCD(b,q)
retorna $(d, l, k - lr)$

Algoritmo 10.9 Algoritmo estendido de Euclides.

a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0
r	1	1	1	2	2	5	
i	12	–7	5	–2	1	0	1
j	–19	12	–7	5	–2	1	0

Tabela 10.10 Execução de ExtendedEuclidGCD(a, b) para $a = 412$ e $b = 260$ para determinar (d, i, j) tais que $d = \gcd(a, b) = ia + jb$. Para cada chamada recursiva, mostramos os argumentos a e b, a variável r e os valores de saída i e j. O valor de saída d é sempre gcd(412, 260) = 4.

10.1.6 Testes de primalidade

Os números primos desempenham um papel importante no processamento numérico, incluindo aplicações criptográficas. Mas como podemos testar se um número n é primo, especialmente se ele for grande?

Testar todos os divisores de n é computacionalmente inviável para n grande. Alternativamente, o pequeno teorema de Fermat (Teorema 10.8) parece sugerir uma solução eficiente. Talvez possamos usar a equação

$$a^{p-1} \equiv 1 \bmod p$$

para testar p. Ou seja, podemos escolher um número a e elevá-lo à potência $p - 1$ módulo p. Se o resultado ***não*** for 1, então o número p não é primo; caso contrário, existe uma chance de que ele o seja. Repetir esse teste para vários valores de a provaria que p é primo? A resposta infelizmente é "não". Existe uma classe de números, chamados de ***números de Carmichael***, que tem a propriedade $a^{n-1} \equiv 1 \bmod n$ para todo $1 \leq a \leq n - 1$, mas n é composto. A existência desses números acaba com o teste simples proposto acima. Os números 561 e 1105 são exemplos de números de Carmichael.

Um padrão para testes de primalidade

Mesmo que o teste probabilístico acima não funcione, existem vários testes relacionados que funcionam fazendo uso mais sofisticado do pequeno teorema de Fermat. Esses testes probabilísticos de primalidade são baseados na abordagem geral a seguir: seja n um inteiro ímpar que desejamos testar, e seja witness(x, n) uma função booleana de uma variável aleatória x e n com as seguintes propriedades:

1. Se n é primo, então witness(x, n) sempre é falsa. Assim, se witness(x, n) é verdadeira, então n é comprovadamente um número composto.
2. Se n é composto, então witness(x, n) é falsa com probabilidade $q < 1$.

A função witness é chamada de ***função testemunha de composição*** com probabilidade de erro q, pois q limita a probabilidade de que witness identificará incorretamente um número composto p como primo. Calculando repetidamente witness(x, n) para valores aleatórios e independentes de x, podemos determinar se n é primo com uma probabilidade de erro arbitrariamente pequena. A probabilidade de que witness(x, n) retorne k "falsos" incorretos para valores independentes de x, quando n é um número composto é q^k. Um algoritmo probabilístico genérico para o teste de primalidade baseado nessa observação é mostrado no Algoritmo 10.11. Esse algoritmo, que é descrito usando um padrão de projeto chamado de método de *templates*, assume que temos uma função testemunha de composição witness que satisfaz as duas condições anteriores. Para transformar o *template* em um algoritmo completo, precisamos apenas especificar os detalhes de como escolher os números aleatórios x e como determinar witness(x, n), a função testemunha de composição.

Algoritmo RandomizedPrimalityTesting(n, k):
  Entrada: inteiro ímpar $n \geq 2$ e parâmetro de confiança k.
  Saída: uma indicação de que n é composto (que está sempre correta) ou é primo (que pode estar incorreta com probabilidade 2^{-k}).
  {Este método assume que temos uma função testemunha de composição witness(x, n) com probabilidade de erro $q < 1$. }
  $t \leftarrow \lceil k/\log_2(1/q) \rceil$
  para $i \leftarrow 1$ até t **faça**
    $x \leftarrow$ random()
    se witness(x, n) **então**
      retorne "composto"
  retorne "primo"

Algoritmo 10.11 Um *template* para um algoritmo probabilístico genérico para teste de primalidade baseado na função testemunha de composição witness. Assumimos que o método auxiliar random() escolhe um valor aleatório no domínio da variável aleatória x.

Se o método RandomizedPrimalityTesting(n, k, witness) retorna "composto", saberemos com certeza que n é composto. Se o método retornar "primo", no entanto, a probabilidade de que n não seja primo não é maior do que 2^{-k}. De fato, suponha que n é composto e o método retorna "primo". Então, a função witness(x, n) foi avaliada como verdadeira para t valores aleatórios de x. A probabilidade desse evento é q^t. Pela relação entre o parâmetro de confiança k, o número de iterações t e a probabilidade de erro q da função witness, temos que $q^t \leq 2^{-k}$.O segundo argumento, k, do método RandomizedPrimalityTesting é um parâmetro de confiança.

O algoritmo de teste de primalidade de Solovay-Strassen

O ***algoritmo de Solovay-Strassen*** para teste de primalidade é uma especialização do método de *templates* RandomizedPrimalityTesting. A função testemunha de composição usada pelo algoritmo é baseada em fatos da teoria dos números, que revisaremos a seguir.

Seja p um número primo ímpar. Um elemento $a > 0$ de Z_p é dito um ***resíduo quadrático*** se ele é o quadrado de algum elemento x de Z_p, ou seja,

$$a \equiv x^2 \pmod{p}$$

Para $a \geq 0$, o ***símbolo de Legendre*** $\left(\frac{a}{p}\right)$ é definido por

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{se } a \bmod p \text{ é um resíduo quadrático} \\ 0 & \text{se } a \bmod p = 0 \\ -1 & \text{em outro caso.} \end{cases}$$

A notação do símbolo de Legendre não deve ser confundida com a operação de divisão. Pode ser mostrado (veja o Exercício C-10.2) que

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}.$$

Generalizamos o símbolo de Legendre removendo a restrição de que p seja primo. Seja n um inteiro positivo ímpar com a decomposição em fatores primos

$$n = p_1^{e_1} \ldots p_k^{e_k}.$$

Para $a \geq 0$, o ***símbolo de Jacobi***,

$$\left(\frac{a}{n}\right),$$

é definido pela equação abaixo:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i} = \left(\frac{a}{p_1}\right)^{e_1} \cdot \left(\frac{a}{p_{2(\ )}}\right)^{e_2} \cdots \left(\frac{a}{p_k}\right)^{e_k}$$

Como o símbolo de Legendre, o símbolo de Jacobi tem valor 0, 1 ou –1. Mostramos no Algoritmo 10.12 um método recursivo para calcular o símbolo de Jacobi. A justificativa de seu funcionamento é omitida (veja o Exercício C-10.5).

Algoritmo Jacobi(a, b):
  Entrada: inteiros a e b.
  Saída: o valor do símbolo de Jacobi $\left(\frac{a}{b}\right)$.
se $a = 0$ **então**
  retorne 0
senão se $a = 1$ **então**
  retorne 1
senão se a mod 2 = 0 **então**
  se $(b^2 - 1)/8$ mod 2 = 0 **então**
    retorne Jacobi(a/2, b)
  senão
    retorne – Jacobi(a/2, b)
senão se $(a - 1)(b - 1)/4$ mod 2 = 0 **então**
  retorne Jacobi(b mod a, a)
senão
  retorne – Jacobi(b mod a, a)

Algoritmo 10.12 Cálculo recursivo do símbolo de Jacobi.

Se n é primo, então o símbolo de Jacobi $\left(\frac{a}{n}\right)$ é igual ao símbolo de Legendre. Assim, para cada elemento a de Z_n, temos

$$\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}, \tag{10.2}$$

quando n é primo. Se n é composto, podem existir valores de a que ainda satisfaçam a Equação 10.2. Assim, se a Equação 10.2 é satisfeita, dizemos que n é um ***pseudoprimo Euleriano*** com base a. O lema a seguir, mostrado sem prova, apresenta uma propriedade dos pseudoprimos Eulerianos que fornece uma função testemunha de primalidade.

Lema 10.14: *seja n um número composto. Existem no máximo $(n - 1)/2$ valores positivos de a em Z_n tais que n é um pseudoprimo Euleriano com base a.*

O algoritmo de teste de primalidade de Solovay-Strassen usa a função testemunha de primalidade abaixo:

$$\text{witness}(x,n) = \begin{cases} \textit{falso} & \text{se } n \text{ é um pseudoprimo Euleriano na base } x \\ \textit{verdadeiro} & \text{em outro caso,} \end{cases}$$

onde x é um inteiro aleatório e $1 < x \leq n - 1$. Pelo Lema 10.14, essa função tem probabilidade de erro $q \leq 1/2$. O algoritmo de teste de primalidade de Solovay-Strassen pode ser expresso como uma especialização do método de *templates* RandomizedPrimalityTesting (Algoritmo 10.11) que redefine os métodos auxiliares witness(x, n) e random() como mostrado no Algoritmo 10.13.

Algoritmo witness(x, n):
retorna (Jacobi(x, n) mod n) $\neq$ FastExponentiation $\left(x, \frac{n-1}{2}, n\right)$

Algoritmo random():
retorna um inteiro aleatório entre 1 e $n - 1$

Algoritmo 10.13 O algoritmo de Solovay-Strassen obtido especializando-se os métodos auxiliares do algoritmo RandomizedPrimalityTesting (Algoritmo 10.11).

A análise do tempo de execução do algoritmo de Solovay-Strassen é simples. Como a probabilidade de erro da função testemunha não é maior do que 1/2, podemos fazer $q = 2$, o que implica que o número de iterações seja igual ao parâmetro de confiança k. A cada iteração, a determinação de witness(x, n) custa $O(\log(n))$ operações aritméticas (veja o Teorema 10.11 e o Exercício C-10.5). Concluímos a seguir.

Teorema 10.15: *dado um inteiro positivo ímpar n e um parâmetro de confiança $k > 0$, o algoritmo de Solovay-Strassen determina se n é primo com probabilidade de erro 2^{-k} realizando $O(k \log(n))$ operações aritméticas.*

O algoritmo de teste de primalidade de Rabin-Miller

Agora descreveremos o ***algoritmo de Rabin-Miller*** para teste de primalidade. Esse algoritmo é baseado no pequeno teorema de Fermat (Teorema 10.8) e no lema a seguir.

Lema 10.16: *seja $p > 2$ um número primo. Se x é um elemento de Z_p tal que*

$$x^2 \equiv 1 \pmod{p},$$

então

$$x \equiv 1 \pmod{p}$$

ou

$$x \equiv -1 \pmod{p}.$$

Uma ***raiz quadrada não-trivial da unidade*** em Z_n é definida como um inteiro $1 < x < n - 1$ tal que

$$x^2 \equiv 1 \pmod{n},$$

O Lema 10.16 afirma que, se n é primo, então não existem raízes quadradas não-triviais da unidade em Z_n.

Para um inteiro ímpar n, seja a representação binária de $n - 1$

$$r_{b-1} r_{b-2} \ldots r_1 r_0.$$

Defina s_i como o número cuja representação binária é dada pelos i *bits* mais à esquerda em $n - 1$, ou seja, s_i é escrito em binário como

$$r_{b-1} \ldots r_{b-i}.$$

Dado um inteiro x, defina o elemento X_i de Z_n como

$$X_i = x^{S_i} \bmod n.$$

O algoritmo de Rabin-Miller define sua função testemunha witness(x, n) como verdadeira se e somente se $x^{n-1} \bmod n \neq 1$. X_i é uma raiz quadrada não-trivial da unidade para algum $1 < i < b - 1$. A computação dessa função é mais fácil do que parece. De fato, se calcularmos $x^{n-1} \bmod n$ usando o algoritmo dos quadrados sucessivos (Algoritmo 10.7), os inteiros X_i são um subproduto da computação (veja o Exercício C-10.6). A probabilidade de erro é dada pelo lema a seguir, mostrado sem prova.

Lema 10.17: *seja n um número composto. Existem no máximo (n – 1)/4 valores positivos de x em Z_n tal que a função testemunha do algoritmo de Rabin-Miller* witness*(x, n) retorna "true".*

Concluímos como segue.

Teorema 10.18: *dados um inteiro positivo ímpar n e um parâmetro de confiança $k > 0$, o algoritmo de Rabin-Miller determina se n é primo com probabilidade de erro 2^{-k} usando $O(k \log(n))$ operações aritméticas.*

O algoritmo de Rabin-Miller é muito usado na prática para testes de primalidade.

Encontrando números primos

Um algoritmo de teste de primalidade pode ser usado para selecionar um número primo aleatório em uma dada faixa de valores ou com um número de *bits* preestabelecido. Usamos o resultado a seguir, obtido a partir da teoria dos números e mostrado sem prova.

Teorema 10.19: *o número $\pi(n)$ de números primos menores ou iguais a n é $\Theta(n/\ln(n))$. De fato, se $n \geq 17$, então $n/\ln(n) < \pi(n) < 1{,}26n/\ln(n)$.*

No teorema acima, $\ln(n)$ é o logaritmo natural de n, ou seja, o logaritmo de n na base e onde e é o número de Euler, um número transcendente cujos primeiros dígitos são 2,71828182845904523536...

Uma conseqüência do Teorema 10.19 é que um inteiro aleatório n é primo com probabilidade $1/\ln(n)$. Assim, para encontrar um número primo com um dado número b de *bits,* geramos números ímpares aleatórios de b *bits* e eles são testados por primalidade até que encontremos um número primo.

Teorema 10.20: *dado um inteiro b e um parâmetro de confiança k, um número primo aleatório com b bits pode ser escolhido com probabilidade de erro 2^{-k} realizando $O(kb)$ operações aritméticas.*

10.2 Processos criptográficos

A Internet está possibilitando que um crescente número de atividades, como a troca de correspondência, o comércio eletrônico e as transações financeiras, sejam realizadas eletronicamente. Entretanto, a própria Internet é uma rede insegura de transmissão de informações: os dados transmitidos pela Internet passam por vários computadores intermediários chamados de ***roteadores***, que podem acessar os dados e potencialmente modificá-los.

Uma variedade de técnicas criptográficas foi desenvolvida para suportar a comunicação segura usando uma rede insegura como a Internet. Em particular, a pesquisa criptográfica desenvolveu os seguintes processos criptográficos úteis:

- ***Encriptação/decriptação:*** uma mensagem M a ser transmitida, chamada de ***texto***, é transformada em uma seqüência ilegível de caracteres, chamada de ***texto cifrado***, antes de ser mandada pela rede. Essa transformação é chamada de ***encriptação***. Depois que o texto cifrado C é recebido, ele é reconvertido para o texto M usando a transformação inversa (que depende de informação secreta adicional). Esse processo reverso é chamado de ***decriptação***. Um ingrediente essencial da encriptação é que, para alguém diferente do destinatário, deve ser computacionalmente inviável transformar C em M (sem saber a informação secreta de decriptação conhecida pelo destinatário).
- ***Assinatura digital:*** o autor da mensagem M computa uma mensagem S derivada de M e de informações secretas conhecidas apenas por ele próprio. A mensagem S é uma ***assinatura digital*** se outra pessoa puder verificar facilmente que somente o autor de M poderia ter determinado S em um tempo razoável.

Uso de técnicas criptográficas para serviços de segurança de informação

A encriptação e as assinaturas digitais são, por vezes, combinadas com outros processos criptográficos, alguns dos quais discutiremos adiante. Ainda assim, as duas técnicas acima já são suficientes para suportar os serviços de segurança de informação discutidos na introdução:

- ***Integridade de dados:*** calcular a assinatura digital S de uma mensagem M não apenas ajuda a determinar o autor de M, mas também verifica a integridade de M, pois uma modificação de M produziria uma assinatura diferente. Assim, para realizar uma verificação de integridade, podemos realizar um teste de que S é realmente uma assinatura para a mensagem M.
- ***Autenticação:*** as técnicas criptográficas anteriores podem ser usadas para realizar a autenticação de duas formas. Em esquemas de autenticação com ***senha***, um usuário fornece sua identificação e uma senha para uma aplicação cliente. Ambas são encriptadas imediatamente e mandadas para uma aplicação que as autenticará. Se a identificação e a senha encriptadas combinam com as armazenadas em um banco de dados, então o indivíduo é autenticado (note que a aplicação da autenticação guarda os dados sempre em formato criptografado). Alternativamente, um autenticador pode lançar um desafio a um usuário sob a forma de uma mensagem qualquer M, a qual o usuário deve poder assinar digitalmente para ter sua autenticação.
- ***Autorização:*** dado um sistema de autenticação, podemos instalar autorizações mantendo ***listas de controle de acesso*** que são associadas a dados sigilosos ou a programas que podem ser acessados somente por indivíduos autorizados. Alternativamente o usuário com acesso a dados sigilosos pode assinar digitalmente uma mensagem C que autoriza um usuário a realizar certas tarefas. Por exemplo, a mensagem poderia ser "O diretor financeiro deve dar à pessoa x acesso aos dados financeiros do primeiro semestre''.

- ***Confidencialidade:*** informações sigilosas podem ser mantidas secretas por encriptação.
- ***Irrefutabilidade:*** se as partes estão negociando um contrato M e assinam a mensagem digitalmente, então temos uma forma de provar que elas conhecem e aceitam o conteúdo da mensagem M.

Esta seção dá uma introdução às tecnicas criptográficas. Nomes de pessoas, como Alice, Bob e Eva, são usados para denotar as partes envolvidas no protocolo criptográfico. Concentramo-nos no esquema de ***criptografia de chave pública***, que é baseado em propriedades da teoria dos números e algoritmos discutidos na seção anterior. Antes de introduzir os conceitos da criptografia de chave pública, discutiremos brevemente uma abordagem alternativa para a encriptação.

10.2.1 Esquemas simétricos de encriptação

Como dito anteriormente, um problema fundamental da criptografia é a confidencialidade, ou seja, mandar uma mensagem de Alice para Bob de forma que uma terceira pessoa, Eva, não obtenha nenhuma informação se interceptar uma cópia da mensagem. Além disso, observamos que a confidencialidade pode ser obtida pela ***encriptação*** ou ***cifragem***, em que a mensagem M a ser transmitida, chamada de ***texto***, é codificada em uma seqüência ilegível de caracteres C, chamada de ***texto cifrado***, antes de ser mandada pela rede. Depois que o texto cifrado C é recebido, ele é decodificado usando a transformação inversa, e o texto M é obtido.

Chaves secretas

Ao descrever os detalhes de um esquema de criptografia, devemos explicar todos os passos necessários para codificar um texto M em um texto cifrado C e a forma como decodificar C para obter M outra vez. Além disso, para que Eva seja incapaz de obter M a partir de C, deve haver informações secretas que sejam mantidas escondidas de Eva.

Na criptografia tradicional, uma ***chave secreta*** k é compartilhada entre Alice e Bob e usada por ambos para codificar e decodificar a mensagem. Tais esquemas são chamados de ***encriptação simétrica***, pois a chave k é usada tanto para encriptação como para decriptação e compartilhada igualmente entre Alice e Bob.

Métodos de substituição

Um exemplo clássico de encriptação simétrica é um sistema de ***substituição***, em que a chave secreta é uma permutação π dos caracteres no alfabeto. Codificar o texto M e obter o texto cifrado C consiste em substituir cada caracter x de M pelo caracter $y = \pi(x)$. A decodificação pode ser feita facilmente conhecendo-se a função de permutação π. De fato, M é derivado de C substituindo-se cada caracter de C pelo caracter $x = \pi^{-1}(y)$. O ***código de César*** é um antigo exemplo de método de substituição, em que cada caracter x é substituído pelo caracter

$$y = x + k \bmod n,$$

onde n é o tamanho do alfabeto e $1 < k < n$ é a chave secreta. Este esquema de substituição é conhecido como o código de César, pois Júlio César a usava com $k = 3$.

Métodos de substituição são fáceis de usar, mas não são seguros. Na verdade, a chave secreta pode ser facilmente obtida usando uma ***análise de freqüência*** e o conhecimento da freqüência das diferentes letras ou grupos de letras consecutivas na linguagem usada para o texto.

O bloco de uso único

Existem esquemas simétricos que são seguros. De fato, o esquema mais seguro conhecido é um esquema simétrico conhecido como o ***bloco de uso único***. Neste esquema, Alice e Bob compartilham uma seqüência aleatória de *bits* K tão grande quanto qualquer mensagem que eles queiram transmitir. Essa seqüência K é a chave simétrica, pois, para obter um texto cifrado C de uma mensagem M, Alice faz a operação

$$C = M \oplus K,$$

onde "$\oplus$" denota o operador de "ou exclusivo" *bit* a *bit*. Ela pode então mandar C para Bob usando qualquer meio de comunicação confiável, até mesmo um em que Eva esteja espionando, pois o texto cifrado C não pode ser diferenciado de uma seqüência aleatória de *bits*. Ainda assim, Bob pode facilmente obter a mensagem M a partir de C computando $C \oplus K$, pois

$$\begin{aligned} C \oplus K &= (M \oplus K) \oplus K \\ &= M \oplus (K \oplus K) \\ &= M \oplus \mathbf{0} \\ &= M, \end{aligned}$$

onde $\mathbf{0}$ denota a seqüência de *bits* que é totalmente composta de zeros do mesmo tamanho de M. Esse esquema é claramente simétrico, pois a chave K é usada para encriptação e decriptação.

O bloco de uso único é computacionalmente eficiente, pois a operação de "ou exclusivo" é uma das mais rápidas realizáveis em um computador. Além disso, como já foi mencionado, o bloco de uso único é muito seguro. Mesmo assim, ele não é muito usado, pois seu problema principal é que Alice e Bob devem compartilhar uma chave muito grande. Além disso, a segurança do bloco de uso único depende do fato de que a chave K é usada apenas uma vez. Se K for reutilizada, existem várias análises criptográficas simples que podem quebrar o esquema. Para criptografia na prática, preferimos chaves secretas que possam ser reutilizadas e sejam menores do que as mensagens que elas codificam.

Outros esquemas simétricos

Existem sistemas simétricos que são seguros e eficientes. Esses sistemas são denominados por siglas ou nomes como "3DES", "IDEA", "Blowfish" e "Rijndael", que realizam uma seqüência de transformações complexas de substituição e permutação nos *bits* do texto. Embora esses algoritmos sejam importantes em várias aplicações, eles são menos interessantes de um ponto de vista algorítmico e estão fora do escopo deste livro. Eles são executados em tempo proporcional ao comprimento da mensagem sendo codificada ou decodificada; portanto, mencionamos que esses algoritmos existem e são rápidos, mas neste livro não discutiremos nenhum desses esquemas simétricos eficientes em mais detalhe.

10.2.2 Esquemas de chave pública

Um grande problema com esquemas simétricos é a ***transferência de chaves***, ou como distribuir a chave secreta usada para codificação. Em 1976, Diffie e Hellman descreveram um sistema abstrato que evitaria esses problemas, o ***esquema de chave pública***. Embora eles não tenham publicado um esquema de chave pública específico, eles discutiram as qualidades de um tal sistema. Especificamente, dada uma mensagem M, uma função de encriptação E e uma função de decriptação D, as quatro propriedades abaixo devem ser válidas:

1. $D(E(M)) = M$.
2. E e D podem ser calculadas facilmente.
3. É computacionalmente inviável[1] obter D a partir de E.
4. $E(D(M)) = M$.

Em retrospecto, essas propriedades parecem de muito bom senso. A primeira propriedade simplesmente diz que, uma vez que uma mensagem é codificada, aplicar a função de decriptação vai restaurá-la. A segunda propriedade é mais óbvia: para um esquema de criptografia ser prático, a encriptação e a decriptação devem ser computacionalmente rápidas.

A terceira propriedade é o começo da inovação. Ela significa que E funciona apenas em um sentido: é computacionalmente inviável inverter E, a não ser que já se conheça D. Assim, o procedimento de encriptação E pode se tornar público. Qualquer pessoa pode mandar uma mensagem cifrada, e somente uma pode decifrá-la.

Se a quarta propriedade for satisfeita, então o mapeamento é único. Assim, o esquema é uma solução para o problema da ***assinatura digital***. Dada uma mensagem eletrônica de Bob para Alice, como podemos provar que Bob realmente a enviou? Bob pode aplicar seu procedimento de decriptação para uma mensagem de assinatura M. Qualquer pessoa pode então verificar que Bob mandou a mensagem aplicando a M o procedimento E de encriptação. Como somente Bob conhece a função de decriptação, somente Bob pode ter gerado a mensagem de assinatura que é corretamente decodificada pela função E.

A criptografia de chave pública é a base da criptografia moderna. Sua importância econômica está crescendo depressa, pois ela fornece a infra-estrutura de segurança para todas transações eletrônicas na Internet.

O projeto de esquemas de chave pública pode ser descrito em termos gerais. A idéia é encontrar um problema muito difícil para a Computação, e de alguma forma amarrar o esquema criptográfico a ele. Em uma situação ideal, pode-se chegar a uma prova de que quebrar o esquema é computacionalmente equivalente a resolver o problema difícil. Existe uma grande classe de problemas, chamados de ***NP-completos***, que não possuem algoritmos de tempo polinomial conhecidos para sua solução (veja o Capítulo 13). De fato, acredita-se que não existam tais algoritmos. Então, para gerar as chaves de encriptação e decriptação, criamos um conjunto particular de parâmetros para o problema. A encriptação então passa a transformar a mensagem em uma instância do problema, e o destinatário pode usar informações secretas (a chave de decriptação) para resolver o problema eficientemente.

[1] O conceito de dificuldade computacional está formalizado no Capítulo 13.

10.2.3 O esquema RSA

Deve-se ter cuidado com a ligação do problema computacionalmente difícil e o esquema de criptografia. Um dos primeiros esquemas de chave pública, o esquema de Merkle-Hellman, ligava a encriptação a algo chamado de problema da mochila, que é ***NP***-completo. Infelizmente, os problemas que o sistema gera são um subconjunto em especial do problema da mochila, que pode ser facilmente solucionado. Assim, projetar esquemas de chave pública tem suas sutilezas.

Provavelmente o esquema de chave pública mais conhecido é também um dos mais antigos e está ligado ao problema de fatorar grandes números. Ele é chamado de ***RSA*** por causa de seus inventores, Rivest, Shamir e Adleman.

Neste esquema, começamos escolhendo dois números primos grandes, p e q. Seja $n = pq$ o seu produto, e lembre que $\phi(n) = (p-1)(q-1)$. As chaves de encriptação e decriptação são escolhidas de forma que

- e e $\phi(n)$ são primos relativos,
- $ed \equiv 1 \pmod{\phi(n)}$.

A segunda condição quer dizer que d é o inverso multiplicativo de e mod $\phi(n)$. O par de valores n e e forma a chave pública, enquanto d é a chave privada. Na prática, e é escolhido aleatoriamente ou como um dos seguintes números: 3, 17 ou 65537.

As regras para encriptação e decriptação no RSA são simples. Vamos assumir, para simplificar, que o texto é um inteiro M com $0 < M < n$. Se M é uma seqüência de caracteres, podemos ver M como um inteiro concatenando os *bits* de seus caracteres. O texto M é codificado produzindo o texto cifrado C com uma exponenciação modular usando a chave de encriptação e como expoente:

$$\boxed{C \leftarrow M^e \bmod n \text{ (encriptação RSA).}}$$

A decriptação do texto cifrado C também é realizada com exponenciação, usando agora a chave de decriptação d como expoente:

$$\boxed{M \leftarrow C^d \bmod n \text{ (decriptação RSA).}}$$

O funcionamento dessas regras para encriptação e decriptação é justificado pelo teorema a seguir.

Teorema 10.21: *sejam p e q dois números primos ímpares e defina $n = pq$. Seja e um número relativamente primo a $\phi(n)$ e seja d o inverso multiplicativo de e módulo $\phi(n)$. Para cada inteiro x tal que $0 < x < n$,*

$$x^{ed} \equiv x \pmod{n}.$$

Prova: seja $y = x^{ed} \bmod n$. Desejamos provar que $y = x$. Pela forma com que escolhemos e e d, podemos escrever $ed = k\phi(n) + 1$ para algum inteiro k. Assim, temos

$$y = x^{k\phi(n)+1} \bmod n.$$

Distinguimos dois casos.

Caso 1: x não divide n. Reescrevemos então

$$\begin{aligned} y &= x^{k\phi(n)+1} \bmod n \\ &= xx^{k\phi(n)} \bmod n \\ &= x(x^{\phi(n)} \bmod n)^k \bmod n. \end{aligned}$$

Pelo Teorema 10.9 (teorema de Euler), temos $x^{\phi(n)} \bmod n = 1$, o que implica $y = x \cdot 1^k \bmod n = x$.

Caso 2: x divide n. Como $n = pq$, com p e q primos, x é um múltiplo de p ou de q. Suponha que x é um múltiplo de p, ou seja, $x = hp$ para algum inteiro positivo h. É claro que x não pode ser também um múltiplo de q, pois, se fosse, x seria maior do que $n = pq$, uma contradição. Assim, $\gcd(x, q) = 1$ e, pelo Teorema 10.9 (teorema de Euler), temos

$$x^{\phi(q)} \equiv 1 \pmod q.$$

Como $\phi(n) = \phi(p)\,\phi(q)$, elevando ambos os lados da congruência acima à potência $k\phi(q)$, obtemos

$$x^{k\phi(n)} \equiv 1 \pmod q,$$

que reescrevemos como

$$x^{k\phi(n)} = 1 + iq,$$

para algum inteiro i. Multiplicando ambos os lados da igualdade acima por x e lembrando que $x = hp$ e $n = pq$, obtemos

$$\begin{aligned} x^{k\phi(n)+1} &= x + xiq \\ &= x + hpiq \\ &= x + (hi)n. \end{aligned}$$

Assim, temos

$$y = x^{k\phi(n)+1} \bmod n = x.$$

Em qualquer caso, mostramos que $y = x$, o que conclui a prova do teorema. ■

Uso do RSA para assinaturas digitais

A simetria das funções de encriptação e decriptação implica que o esquema RSA suporta assinaturas digitais imediatamente. De fato, uma assinatura digital S para a mensagem M é obtida aplicando a função de decriptação em M, ou seja,

$$S \leftarrow M^d \bmod n \text{ (assinatura RSA).}$$

A verificação da assinatura digital S é agora realizada com a função de encriptação, ou seja, verificando que

$$M \equiv S^e \pmod{n} \text{ (verificação RSA).}$$

A dificuldade de quebrar o RSA

Note que mesmo se conhecermos o valor de e não conseguimos descobrir d a não ser que conheçamos $\phi(n)$. A maior parte dos pesquisadores em criptografia acredita que quebrar o RSA exige calcular $\phi(n)$, e isso significa fatorar n. Mesmo não havendo ***prova*** de que a fatoração é computacionalmente difícil, gerações inteiras de matemáticos famosos têm trabalhado no problema nas últimas centenas de anos. Se n é grande ($\approx$ 200 dígitos), levará muito tempo para ser fatorado. Para dar uma idéia do estado da arte, os matemáticos ficaram bastante empolgados quando uma rede de computadores espalhados pelos Estados Unidos foi capaz de fatorar o nono número de Fermat, $2^{512} - 1$. Este número tem "apenas" 155 dígitos decimais. A não ser que aconteça uma descoberta fundamental, o sistema RSA permanecerá seguro. Se a tecnologia avançar a um ponto em que seja possível fatorar números de 200 dígitos, precisamos apenas escolher um n com 300 ou 400 dígitos.

Análise e inicialização do sistema RSA

O tempo de execução do esquema RSA de encriptação, decriptação, assinatura e verificação é fácil de analisar. Na verdade, cada uma dessas operações requer um número constante de exponenciações modulares que podem ser realizadas com o método FastExponentiation (Algoritmo 10.7).

Teorema 10.22: *seja n o módulo usado no esquema RSA. Então a encriptação, a decriptação, a assinatura e a verificação custam* $O(\log(n))$ *operações aritméticas.*

Para iniciar o esquema RSA, precisamos gerar o par de chaves; a pública e a privada. Precisamos calcular a chave privada (d, p, q) e a chave pública (e, n) que a acompanha. Isso envolve os seguintes cálculos:

- Escolha de dois números primos aleatórios p e q com um dado número de *bits*. Isso pode ser feito testando a primalidade de números inteiros, como foi discutido no fim da Seção 10.1.6.
- Escolha de um inteiro e relativamente primo a $\phi(n)$. Isso pode ser feito escolhendo primos aleatórios menores do que $\phi(n)$ até encontrarmos um que não divida $\phi(n)$. Na prática, é suficiente verificar primos pequenos de uma lista de primos conhecidos (muitas vezes $e = 3$ e $e = 17$ funcionam).
- Cálculo do inverso multiplicativo d de e em $Z_{\phi(n)}$. Isso pode ser feito usando o algoritmo estendido de Euclides (Corolário 10.13).

Explicamos previamente neste capítulo os algoritmos para cada um desses problemas de teoria dos números.

10.2.4 O esquema El Gamal

Vimos que a segurança do esquema RSA é relacionada com a dificuldade de fatorar números grandes. É possível construir esquemas baseados em outros problemas difíceis da teoria dos números. Agora consideramos o esquema El Gamal, batizado com o nome de seu inventor, Taher El Gamal, que é baseado na dificuldade de um problema chamado de "logaritmo discreto".

O logaritmo discreto

Quando trabalhamos com números reais, $\log_b y$ é o valor x tal que $b^x = y$. Podemos definir um logaritmo discreto análogo: dados os inteiros b e n com $b < n$, o ***logaritmo discreto*** de um inteiro y na base b é o inteiro x tal que

$$b^x \equiv y \bmod n.$$

O logaritmo discreto também é chamado de ***índice***, e escrevemos

$$x = \mathrm{ind}_{b,\,n} y.$$

Mesmo sendo eficiente elevar números a potências grandes em módulo p (lembre-se do algoritmo dos quadrados sucessivos, Algoritmo 10.7), a computação inversa do logaritmo discreto é muito mais difícil. O esquema El Gamal se baseia na dificuldade desse problema.

Codificação El Gamal

Seja p um primo e g um gerador de Z_p. A chave privada x é um inteiro entre 1 e $p - 2$. Seja $y = g^x \bmod p$. A chave pública para a codificação El Gamal é a tripla (p, g, y). Se calcular logaritmos discretos é tão difícil quanto se acredita, tornar público $y = g^x \bmod p$ não revela o valor de x.

Para fazer a encriptação de um texto M, um inteiro aleatório k relativamente primo a $p - 1$ é escolhido, e o par de valores abaixo é calculado:

$$\boxed{\begin{aligned} a &\leftarrow g^k \bmod p \\ b &\leftarrow My^k \bmod p \end{aligned} \quad \text{(encriptação El Gamal)}}$$

O texto cifrado C consiste no par (a, b) avaliado acima.

Decodificação El Gamal

A decriptação do texto cifrado $C = (a, b)$ no esquema El Gamal, para recuperar o texto M, é simples:

$$\boxed{M \leftarrow b \,/\, a^x \bmod p \text{ (decriptação El Gamal).}}$$

Na expressão acima, a "divisão" por a^x deve ser interpretada no contexto da aritmética modular, ou seja, M é multiplicado pelo inverso de a^x em Z_p. O funcionamento do esquema El Gamal é fácil de verificar. Temos

$$\begin{aligned} b / a^x \mod p &= My^k\left(a^x\right)^{-1} \mod p \\ &= Mg^{xk}\left(g^{kx}\right)^{-1} \mod p \\ &= M. \end{aligned}$$

Uso do El Gamal para assinaturas digitais

Uma variação do esquema acima fornece uma assinatura digital: uma assinatura para a mensagem M é um par $S = (a, b)$ obtido escolhendo-se um inteiro aleatório k relativamente primo a $p - 1$ (que é igual a $\phi(p)$) e calculando

$$\boxed{\begin{aligned} a &\leftarrow g^k \bmod p \\ b &\leftarrow k^{-1}(M - xa) \bmod (p-1) \end{aligned} \qquad \text{(assinatura El Gamal)}}$$

Para verificar uma assinatura digital $S = (a, b)$, verificamos que

$$\boxed{y^a a^b \equiv g^M \pmod p \text{ (verificação El Gamal).}}$$

O funcionamento do esquema de assinatura digital El Gamal pode ser entendido como segue:

$$\begin{aligned} y^a a^b \bmod p &= \left(\left(g^x \bmod p\right)^a \bmod p\right)\left(\left(g^k \bmod p\right)^{k^{-1}(M-xa) \bmod (p-1)} \bmod p\right) \\ &= g^{xa} g^{kk^{-1}(M-xa) \bmod (p-1)} \bmod p \\ &= g^{xa+M-xa} \bmod p \\ &= g^M \bmod p. \end{aligned}$$

Análise do esquema El Gamal

A análise do desempenho do esquema El Gamal é similar à do RSA, pois temos o seguinte teorema:

Teorema 10.23: *seja n o módulo usado no esquema El Gamal. As operações de encriptação, decriptação, assinatura digital e verificação custam, cada uma, $O(\log(n))$ operações aritméticas.*

10.3 Algoritmos e protocolos para segurança de informações

Com as primeiras ferramentas, como os algoritmos fundamentais para computações envolvendo números e os esquemas de criptografia de chave pública, podemos começar a combiná-las com outros algoritmos para fornecer serviços de segurança de informação. Discutiremos vários desses protocolos nesta seção, muitos dos quais usando, além dos algoritmos discutidos anteriormente, o tópico que discutiremos a seguir.

10.3.1 Funções *hash* unidirecionais

Sistemas de criptografia de chave pública são muitas vezes usados junto com uma ***função hash unidirecional***, também chamada de ***resumo da mensagem*** ou ***impressão digital***. Fornecemos a seguir uma descrição informal de uma função desse tipo. Uma discussão formal está fora do escopo deste livro.

Uma ***função hash unidirecional*** H mapeia uma seqüência (mensagem) M de comprimento arbitrário em um inteiro $d = H(M)$ com um número fixo de *bits*, chamado de ***resumo*** de M, satisfazendo às seguintes propriedades:

1. Dada uma seqüência M, o resumo de M pode ser determinado rapidamente.
2. Dado o resumo d de M, mas não M, é computacionalmente inviável determinar M.

Uma função *hash* unidirecional é dita ***resistente a colisões*** se, dada uma seqüência M, for computacionalmente inviável encontrar outra seqüência M' com o mesmo resumo, e é dita ***fortemente resistente a colisões*** se for computacionalmente inviável encontrar duas seqüências M_1 e M_2 com o mesmo resumo.

Várias funções *hash* unidirecionais consideradas fortemente resistentes a colisões já foram projetadas. As mais usadas na prática são a MD5, que produz um resumo de 128 *bits*, e a SHA-1, que produz um resumo de 160 *bits*. Examinaremos agora algumas aplicações de uma função *hash* unidirecional.

A primeira aplicação das funções *hash* unidirecionais é acelerar o processo de construção das assinaturas digitais. Se tivermos uma função *hash* unidirecional resistente a colisões, podemos assinar o resumo da mensagem em vez de assinar a própria mensagem, ou seja, a assinatura S é dada por

$$S = D(H(M)).$$

Exceto para mensagens pequenas, aplicar a função sobre a mensagem e depois assinar o resumo é mais rápido do que assinar a mensagem diretamente. Esse procedimento também supera uma restrição importante dos esquemas de assinatura RSA e El Gamal, que exigem que a mensagem M seja menor do que o módulo n. Por exemplo, quando usamos MD5, podemos assinar mensagens de comprimento arbitrário usando um módulo fixo n que deve ser apenas maior do que 2^{128}.

10.3.2 Carimbos de tempo e dicionários autenticados

A próxima aplicação são os ***carimbos de tempo***. Suponha que Alice tenha um documento M e deseje obter uma certificação de que o documento M existe no tempo presente t.

Em um método de carimbo de tempo (também chamado de registro de tempo ou *time stamp*), Alice usa os serviços de uma terceira parte confiável, Trevor, que fornece o carimbo. Alice pode mandar M para Trevor e pedir que ele assine um novo documento M', que é a concatenação de M e t. Embora essa abordagem funcione, ela tem a desvantagem de que Trevor pode ver M. Uma função *hash* unidirecional e resistente a colisões H pode eliminar esse problema: Alice determina o resumo d de M usando H e pede a Trevor que assine uma nova mensagem M'', que consiste na concatenação de d e t.

Dicionários autenticados

Na verdade, podemos definir outro método para o protocolo de tempo que não requer que confiemos completamente em Trevor. Esse esquema alternativo é baseado no conceito de um ***dicionário autenticado***.

Em um dicionário autenticado, a terceira parte, Trevor, guarda um dicionário de itens. Na aplicação de carimbo de tempo, os itens são os resumos das mensagens que precisam ser carimbadas e comprovadas em certa data. Neste caso, no entanto, não confiamos na afirmação de Trevor (feita através da assinatura) de que um resumo d do documento M de Alice existe em um certo tempo t. Em vez disso, Trevor determina um resumo D do dicionário inteiro e torna D público em um local onde o carimbo de tempo não possa ser questionado (como a seção de classificados de um jornal). Além disso, Trevor responde a Alice com um resumo parcial D' que sumariza todos os itens no dicionário, exceto o resumo d do documento M de Alice.

Para que esse esquema funcione efetivamente, deve existir uma função f tal que $D = f(D', d)$, com f sendo fácil de calcular (para Alice). Mas essa função deve ser unidirecional no sentido de que, dado um y arbitrário, deve ser computacionalmente difícil (para Trevor) determinar um x tal que $D = f(x, y)$. Dada uma função desse tipo, podemos confiar em Trevor para calcular os resumos de todos os documentos que ele recebe e publicar os resumos em um lugar público. Será computacionalmente inviável para Trevor fabricar uma resposta indicando que um valor d estava no dicionário quando, de fato, ele não estava. Assim, o componente-chave deste protocolo é a existência de uma função unidirecional f. No restante desta subseção, exploraremos um possível método para construir uma função f adequada para um dicionário autenticado.

Árvores *hash*

Uma abordagem interessante atráves de estruturas de dados, chamada de esquema das ***árvores hash***, pode ser usada para implementar um dicionário autenticado. Essa estrutura suporta a construção inicial do dicionário, seguindo-se de operações de consulta ou respostas sobre a existência de itens.

Uma árvore *hash* T para um conjunto S armazena os elementos de S nos nodos externos de uma árvore binária completa T e um valor de *hash* $h(v)$ a cada nodo v, que se combina com o *hash* de seus filhos através de uma função *hash* unidirecional conhecida. Na aplicação de carimbo de tempo, os itens armazenados nos nodos externos de T são eles próprios os resumos de documentos que devem ser garantidos como existentes em um dado momento. O dicionário autenticado para S consiste na árvores *hash* T mais a publicação do valor $h(r)$ armazenado na raiz r de T. Prova-se que um elemento x pertence a S relatando-se os valores armazenados nos nodos do caminho de r a x em T mais os valores de todos os nodos que tenham irmãos nesse caminho.

Dado um tal caminho p, Alice pode recalcular o valor de *hash* $h(r)$ da raiz. Como T é uma árvore binária completa, ela precisa fazer apenas $O(\log(n))$ chamadas à função h para calcular esse valor, onde n é o número de elementos em S.

10.3.3 Lançamento de moedas e compromisso de *bits*

Apresentamos agora um protocolo que permite que Alice e Bob ***lancem uma moeda aleatória*** pela troca de mensagens eletrônicas ou outro tipo de comunicação por uma

rede. Seja H uma função *hash* unidirecional fortemente resistente a colisões. A interação entre Alice e Bob consiste nos passos a seguir:

1. Alice escolhe um número x e calcula o resumo $d = H(x)$ mandando d para Bob.
2. Depois de receber d, Bob manda para Alice sua adivinhação se x é par ou ímpar.
3. Alice anuncia o resultado do lançamento da moeda: se Bob adivinhou corretamente o resultado, é cara; se Bob errou o resultado, é coroa. Ela também manda x para Bob como prova do resultado.
4. Bob verifica que Alice não o enganou, confirmando que $d = H(x)$.

Ser fortemente resistente a colisões é fundamental, pois, se não fosse assim, Alice poderia escolher dois números, um par e outro ímpar, com o mesmo resumo d, sendo capaz de controlar o resultado do lançamento da moeda.

O ***compromisso de bits*** está relacionado ao lançamento de moedas. Neste caso, Alice deseja se comprometer com um valor n (que pode ser um único *bit* ou uma seqüência de *bits*) sem revelar n a Bob. Quando Alice finalmente revelar n, Bob desejará se certificar de que ela não o enganou. Por exemplo, Alice pode desejar provar a Bob que amanhã uma certa ação de mercado vai subir ($n = 1$), cair ($n = -1$) ou se manter ($n = 0$). Usando uma função *hash* unidirecional fortemente resistente a colisões H, o protocolo é este:

1. Ela manda a Bob x mais o resumo da concatenação de x, y e n. Para manter a coerência com a literatura de criptografia, denotamos a concatenação de seqüências a e b como "$a||b$" neste capítulo. Assim, usando essa notação, Alice manda a Bob $d = H(x||y||n)$. Note que Bob não pode determinar n a partir de x e d.
2. Ao final das transações do dia seguinte, quando n é publicamente conhecido, Alice manda y a Bob para verificação.
3. Bob verifica que Alice não o enganou, confirmando que $d = H(x||y||n)$.

10.3.4 Protocolo para transações eletrônicas seguras (SET)

Nossa aplicação final é significativamente mais complexa e envolve o uso combinado de encriptação, assinaturas digitais e funções *hash* unidirecionais. Ela é, na verdade, uma versão simplificada do protocolo ***SET***** (transação eletrônica segura) para pagamento seguro com cartões de crédito pela Internet.

Alice deseja comprar um livro de Barney (que tem uma livraria na Internet) usando um cartão de crédito fornecido por Lisa (um banco). Alice está preocupada com sua privacidade: de um lado, ela não quer que Barney conheça seu número de cartão de crédito; por outro lado, ela não quer que Lisa saiba que livro ela comprou de Barney. Entretanto, ela deseja que Barney mande o livro e que Lisa mande o pagamento a Barney. Por fim, Alice também deseja que toda a conversação entre ela, Barney e Lisa seja mantida sigilosa, mesmo que alguém esteja monitorando a Internet. Usando uma função *hash* unidirecional fortemente resistente a colisões, o protocolo é descrito no Algoritmo 10.14.

* N. de R. Do inglês, *secure electronic transaction*.

Propriedades do protocolo SET

As observações a seguir mostram que os requisitos de confidencialidade, integridade, irrecusabilidade e autenticação do protocolo são satisfeitos:

- Barney não pode ver o número de cartão de crédito de Alice, que está armazenado na nota de pagamento P. Barney tem apenas o resumo p de P, e ele não pode determinar P a partir de p, pois H é unidirecional. Barney também tem o texto cifrado C_L de uma mensagem que contém P. Entretanto, ele não pode decodificar C_L, pois ele não tem a chave privada de Lisa.
- Lisa não pode saber qual o livro comprado por Alice, cujo nome é armazenado na ordem de compra O. Lisa tem apenas o resumo o de O, e ela não pode determinar O com o, pois H é unidirecional.
- A assinatura digital S fornecida por Alice serve a um propósito duplo: ela permite que Barney verifique a autenticidade da ordem de compra O de Alice e que Lisa verifique a autenticidade da nota de pagamento P de Alice.
- Alice não pode negar que encomendou o livro especificado na ordem de compra O e que autorizou o pagamento com cartão de crédito pela nota de pagamento P. Na verdade, como H é resistente a colisões, ela não pode falsificar uma ordem de compra ou uma nota de pagamento que tenham os mesmos resumos o e p.
- Toda a comunicação entre as partes usa criptografia de chave pública e garante confidencialidade mesmo na presença de monitoração da rede.

Assim, embora seja um tanto complexo, o protocolo SET ilustra como os métodos criptográficos podem ser compostos para realizar uma operação não-trivial de comércio eletrônico.

1. Alice prepara dois documentos: uma ordem de compra O dizendo que ela deseja comprar o livro de Barney e uma nota de pagamento P dando a Lisa o número de cartão de crédito para a transação e o total a ser pago. Alice determina os resumos

$$o = H(O)$$
$$p = H(P)$$

e produz uma assinatura digital S para o resumo da concatenação de o e p, ou seja,

$$S = D_A(H(o \| p)) = D(H(H(O) \| H(P))),$$

onde D_A é a função usada por Alice para assinar, baseada em sua chave privada. Alice codifica a concatenação de o, P e S com a chave pública de Lisa, o que produz o texto cifrado

$$C_L = E_L(o \| P \| S).$$

Ela também codifica com a chave pública de Barney a concatenação de O, p e S, obtendo o texto cifrado

$$C_B = E_B(O \| p \| S).$$

Ela manda C_L e C_B para Barney.

2. Barney recupera O, p e S decodificando C_B com sua chave privada. Ele verifica a autenticidade da ordem de compra O com a chave pública de Alice verificando que

$$E_A(S) = H(H(O) \| p),$$

e manda C_L para Lisa.

3. Lisa recupera o, P e S decodificando C_L com sua chave privada. Ela verifica a autenticidade da nota de pagamento P com a chave pública de Alice verificando que

$$E_A(S) = H(o \| H(P)),$$

e verifica que P indica um pagamento a Barney. Ela então cria uma mensagem de autorização M que consiste em um número de transação, o nome de Alice e o total a pagar. Lisa determina a assinatura T de M e manda o par (M, T) codificado para Barney com sua chave pública, ou seja, $C_M = E_B(M \| T)$.

4. Barney recupera M e T decodificando C_M e confirma a autenticidade da autorização M com a chave pública de Lisa, verificando que $E_L(T) = \mathrm{M}$. Ele verifica que o nome em M é o de Alice e que o total é o preço correto do livro. Ele completa a ordem mandando o livro para Alice e pede o pagamento a Lisa mandando a ela o número de transação codificado com a chave pública de Lisa.
5. Lisa paga a Barney e cobra a conta de cartão de crédito de Alice.

Algoritmo 10.14 Versão simplificada do protocolo SET.

10.3.5 Distribuição de chaves e seu intercâmbio

Um sistema de criptografia de chave pública presume que as chaves públicas são conhecidas por todas as partes. Por exemplo, se Alice deseja mandar uma mensagem confidencial a Bob, ela precisa conhecer a chave pública de Bob. Além disso, se Alice deseja verificar a assinatura digital de Bob, ela também precisa da chave pública de Bob. Como Alice obtém a chave pública de Bob? Bob pode enviar a chave para Alice, mas se Eva interceptar a comunicação entre Bob e Alice, então Eva pode trocar a chave de Bob por uma chave sua, e enganar Alice de forma que ela revele sua mensagem para Bob ou fazendo-a acreditar que uma mensagem foi assinada por Bob quando, na verdade, foi assinada por Eva.

Certificados digitais

Uma solução para o problema requer a introdução de uma terceira parte, Charlie, que tem a confiança de todos os participantes no protocolo. Também supomos que cada participante tenha a chave pública de Charlie. Charlie envia a cada participante um ***certificado***, que é um atestado assinado digitalmente por Charlie e que contém o nome do participante e sua chave pública. Com isso, Bob manda a Alice o certificado dado a ele por Charlie. Alice extrai do certificado a chave pública de Bob e verifica sua autenticidade usando a chave pública de Charlie (lembre-se da hipótese de que cada participante tem a chave pública de Charlie). Certificados são amplamente usados em aplica-

ções práticas de criptografia de chave pública. Seu formato é descrito no padrão ***X.509*** da ITU (International Telecommunication Union). Além do nome do participante e sua chave pública, um certificado também contém um número serial único e uma data de perda da validade. O fornecedor de um certificado é chamado de ***autoridade certificadora*** (AC, ou CA em inglês).

Em um ambiente realista, os protocolos descritos na seção anterior devem ser modificados introduzindo certificados para a distribuição de chaves públicas. Além disso, os certificados devem ser validados com a chave pública da autoridade certificadora.

Revogação de certificados

Chaves privadas podem ser perdidas, roubadas ou comprometidas de outras maneiras. Quando isso acontece, o CA deve revogar o certificado daquela chave. Por exemplo, Bob pode ter mantido o arquivo com sua chave privada no *laptop*. Se o *laptop* for roubado, Bob deve pedir ao CA que imediatamente revogue seu certificado, pois, de outra forma, o ladrão poderia se passar por Bob. O CA também publica periodicamente uma ***lista de certificados revogados*** (CRL), que é uma lista assinada dos números de série dos certificados revogados dentro de suas validades com um carimbo de tempo.

Quando validar um certificado, o participante deve também obter a CRL mais recente do CA e verificar que o certificado não foi revogado. A idade da CRL (a diferença entre seu carimbo de tempo e a data atual) fornece a medida do risco para o participante que está verificando o certificado. Alternativamente, existem vários métodos *online* para verificar a validade de um certificado digital usando um servidor que mantém as informações de revogação.

Uso de chaves públicas para a troca de chaves simétricas

A criptografia de chave pública supera o problema principal da criptografia de chave simétrica, pois em um sistema de criptografia de chave pública não há necessidade de distribuir chaves secretas antes de iniciar uma comunicação segura. Infelizmente essa vantagem tem um custo, pois os sistemas de chave pública existentes gastam muito mais tempo fazendo a encriptação e a decriptação do que os sistemas de chave simétrica existentes. Assim, na prática os sistemas de criptografia de chave pública são freqüentemente usados com sistemas de criptografia de chave simétrica para superar o desafio do intercâmbio de chaves, a fim de estabelecer um canal de comunicação usando chaves simétricas.

Por exemplo, se Alice deseja estabelecer um canal de comunicação com Bob, ela e Bob podem realizar os seguintes passos:

1. Alice calcula um número aleatório x e a assinatura digital S de x e codifica o par (x, S) usando a chave pública de Bob e mandando o texto cifrado C que foi obtido para Bob.
2. Bob faz a decriptação de C usando a sua chave privada e verifica que foi Alice quem lhe mandou verificando a assinatura S.
3. Bob pode então mostrar a Alice que recebeu x, codificando x com a chave pública de Alice e mandando-o de volta a ela.

A partir deste ponto, eles podem usar o número x como chave secreta em um sistema de criptografia simétrico.

O intercâmbio de chaves secretas de Diffie-Hellman

Se Alice e Bob estiverem se comunicando usando um meio que é confiável, porém não privado, existe outro sistema para determinar uma chave secreta que eles possam usar para futuras comunicações com criptografia simétrica. Esse sistema é o de ***intercâmbio de chaves de Diffie-Hellman***, e consiste nos seguintes passos:

1. Alice e Bob concordam (publicamente) com um número primo grande n e um gerador g em Z_n.
2. Alice escolhe um número aleatório x e manda a Bob $B = g^x \bmod n$.
3. Bob escolhe um número aleatório y e manda a Alice $A = g^y \bmod n$.
4. Alice calcula $K = A^x \bmod n$.
5. Bob calcula $K' = B^y \bmod n$.

Com certeza $K = K'$, e Alice e Bob podem agora usar K (respectivamente K') para comunicação usando um sistema de criptografia simétrico.

10.4 A transformada rápida de Fourier

Um problema comum em muitos sistemas criptográficos é a multiplicação de inteiros muito grandes e de polinômios. A transformada rápida de Fourier é um algoritmo eficiente e surpreendente para multiplicar esses objetos. Descrevemos esse algoritmo para multiplicação de polinômios primeiramente, e depois mostramos como essa abordagem pode ser usada para multiplicar inteiros muito grandes.

Um polinômio representado em ***forma de coeficientes*** é descrito por um vetor de coeficientes $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ como segue:

$$p(x) = \sum_{i=0}^{n-1} a_i x^i,$$

O ***grau*** de um polinômio é o maior índice de um coeficiente não-nulo a_i. Um vetor de coeficientes de comprimento n pode representar polinômios de grau até $n - 1$.

A representação por coeficientes é natural, pois é simples e permite que várias operações polinomiais sejam realizadas rapidamente. Por exemplo, dado um segundo polinômio descrito usando um vetor de coeficientes $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]$ como

$$q(x) = \sum_{i=0}^{n-1} b_i x^i,$$

podemos somar facilmente $p(x)$ e $q(x)$ termo a termo para produzir

$$p(x) + q(x) = \sum_{i=0}^{n-1} (a_i + b_i) x^i.$$

Da mesma forma, a representação por coeficientes de $p(x)$ permite que $p(x)$ seja avaliado eficientemente pela ***regra de Horner*** (Exercício C-1.16) como

$$p(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-2} + xa_{n-1})...)).$$

Assim, com a representação por coeficientes de $p(x)$, podemos somar e avaliar polinômios de grau $n - 1$ em tempo $O(n)$.

A multiplicação de dois polinômios $p(x)$ e $q(x)$, como definido acima, pela forma de coeficientes não é tão simples. Para perceber a dificuldade, considere $p(x)q(x)$:

$$p(x)q(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1 + a_2b_0)x^2 + ... + a_{n-1}b_{n-1}x^{2n-2}.$$

Ou seja,

$$p(x)q(x) = \sum_{i=0}^{2n-2} c_i x^i, \text{ onde } c_i = \sum_{j=0}^{i} a_j b_{i-j}, \text{ para } i = 0,1,...,2n-2.$$

Essa equação define um vetor $\mathbf{c} = [c_0, c_1,\ldots, c_{2n-1}]$, que chamamos de ***convolução*** dos vetores **a** e **b**. Por razões de simetria, vemos a convolução como um vetor de tamanho $2n$, definindo $c_{2n-1} = 0$. Denotamos a convolução de **a** e **b** como $\mathbf{a}*\mathbf{b}$. Se aplicarmos a definição da convolução diretamente, então levará tempo $\Theta(n^2)$ para multiplicar os dois polinômios p e q.

O algoritmo da ***transformada rápida de Fourier*** **(FFT)** permite que realizemos essa multiplicação em tempo $O(n \log(n))$. A melhora da FFT é baseada em uma observação interessante: outra forma de representar um polinômio de grau $n - 1$ é pelo seu valor em n pontos de entrada diferentes. Uma representação assim é única por causa do teorema a seguir.

Teorema 10.24 (teorema de interpolação para polinômios): *dado um conjunto S de n pontos no plano, com* $S = \{(x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_{n-1}, y_{n-1})\}$, *tal que os* x_i *são todos distintos, então existe um único polinômio p(x) de grau* $n - 1$ *com* $p(x_i) = y_i$, *para* $i = 0, 1, ..., n-1$.

Suponha então que podemos representar um polinômio por seu valor em uma coleção de pontos diferentes, e não por seus coeficientes. Esse teorema sugere um método alternativo para multiplicar dois polinômios p e q. Especificamente, avaliamos p e q para $2n$ pontos diferentes $x_0, x_1,\ldots, x_{2n-1}$ e computamos a representação do produto de p e q como o conjunto

$$\{(x_0, p(x_0)q(x_0)), (x_1, p(x_1)q(x_1)), ..., (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\}.$$

Essa computação custaria tempo $O(n)$, dados os $2n$ pares de entradas e saídas para p e q.

O desafio para usar essa abordagem para multiplicar p e q é obter rapidamente os $2n$ pares de entrada e saída para p e q. A aplicação da regra de Horner para $2n$ pontos diferentes levaria tempo $\Theta(n^2)$, que não é assintóticamente mais rápido do que usar a convolução diretamente. Por isso, a regra de Horner não é útil neste caso. É claro que temos liberdade de escolher os $2n$ pontos para os polinômios, e podemos escolher pontos onde eles sejam fáceis de ser avaliados. Por exemplo, $p(0) = a_0$ é um caso simples. Mas temos de escolher um conjunto de $2n$ pontos para avaliar p, e não apenas um ponto. Felizmente o conceito que discutiremos a seguir fornece um conjunto conveniente de pontos que são coletivamente mais fáceis de avaliar do que usando a regra de Horner $2n$ vezes.

10.4.1 Raízes primitivas da unidade

Um número ω é uma ***raiz primitiva da unidade*** para $n \geq 2$, se satisfizer às propriedades a seguir:

1. $\omega^n = 1$, ou seja, ω é uma n-ésima raiz de 1.
2. Os números $1, \omega, \omega^2, ..., \omega^{n-1}$ são diferentes.

Note que essa definição implica que uma n-ésima raiz primitiva da unidade tem um inverso multiplicativo $\omega^{-1} = \omega^{n-1}$, pois

$$\omega^{-1}\omega = \omega^{n-1}\omega = \omega^n = 1.$$

Assim, podemos falar de expoentes negativos para ω, bem como expoentes positivos.

A noção de uma n-ésima raiz primitiva pode parecer uma definição estranha e com poucos exemplos, mas delas existem vários casos importantes. Um destes é o número complexo

$$e^{2\pi i/n} = \cos(2\pi/n) + \mathbf{i}\ \mathrm{sen}(2\pi/n),$$

que é uma n-ésima raiz primitiva da unidade quando realizamos a aritmética com números complexos, onde $\mathbf{i} = \sqrt{-1}$

As n-ésimas raízes primitivas da unidade têm um número de propriedades importantes, incluindo as três a seguir.

Lema 10.25 (propriedade do cancelamento): *se ω é uma n-ésima raiz da unidade, então para qualquer inteiro $k \neq 0$ com $-n < k < n$,*

$$\sum_{j=0}^{n-1} \omega^{kj} = 0$$

Prova: como $\omega^k \neq 1$,

$$\sum_{j=0}^{n-1} \omega^{kj} = \frac{\left(\omega^k\right)^n - 1}{\omega^k - 1} = \frac{\left(\omega^n\right)^k - 1}{\omega^k - 1} = \frac{1^k - 1}{\omega^k - 1} = \frac{1-1}{\omega^k - 1} = 0$$

■

Lema 10.26 (propriedade da redução): *se ω é uma $2n$-ésima raiz primitiva da unidade, então ω^2 é uma n-ésima raiz primitiva da unidade.*

Prova: se $1, \omega, \omega^2, ..., \omega^{2n-1}$ são distintos, então $1, \omega^2, (\omega^2)^2, ..., (\omega^2)^{n-1}$ também são distintos. ■

Lema 10.27 (propriedade reflexiva): *se ω é uma n-ésima raiz da unidade e n é par, então*

$$\omega^{n/2} = -1$$

Prova: pela propriedade do cancelamento, para $k = n/2$,

$$
\begin{aligned}
0 &= \sum_{j=0}^{n-1} \omega^{(n/2)j} \\
&= \omega^0 + \omega^{n/2} + \omega^n + \omega^{3n/2} + \cdots + \omega^{(n/2)(n-2)} + \omega^{(n/2)(n-1)} \\
&= \omega^0 + \omega^{n/2} + \omega^0 + \omega^{n/2} + \cdots + \omega^0 + \omega^{n/2} \\
&= (n/2)(1 + \omega^{n/2}).
\end{aligned}
$$

Assim, $0 = 1 + \omega^{n/2}$. ■

Um corolário interessante da propriedade reflexiva, que motiva seu nome, é o fato de que se ω é uma n-ésima raiz primitiva da unidade e $n \geq 2$ é par, então

$$\omega^{k+n/2} = -\omega^k.$$

10.4.2 A transformada discreta de Fourier

Vamos retornar ao problema de avaliar um polinômio definido por um vetor de coeficientes **a** como

$$p(x) = \sum_{i=0}^{n-1} a_i x^i,$$

para um conjunto de pontos cuidadosamente escolhidos. A técnica que discutiremos na próxima seção é chamada de ***transformada discreta de Fourier (DFT**)***, que avalia $p(x)$ nas n-ésimas raízes da unidade $\omega^0, \omega^1, \omega^2, ..., \omega^{n-1}$. Isso fornece apenas n pares de valores; porém, podemos "completar" nossa representação de coeficientes para p com zeros, fazendo $a_i = 0$ para $n \leq i \leq 2n - 1$. Com isso, veríamos p como um polinômio de grau $2n - 1$, o que, por sua vez, permite que usemos as $2n$-ésimas raízes da unidade como valores para a DFT de p. Assim, se precisarmos de mais valores para avaliação de p, assumimos que o vetor de coeficientes para p já teve adicionados os zeros necessários.

Formalmente, a transformada discreta de Fourier para o polinômio p representado pelo vetor de coeficientes **a** é definida como o vetor **y** de valores

$$y_j = p(\omega^j),$$

onde ω é uma n-ésima raiz primitiva da unidade, ou seja,

$$y_j = \sum_{i=0}^{n-1} a_i \omega^i.$$

Na notação matricial, podemos considerar o vetor **y** de y_j valores, e o vetor **a** como vetores-coluna, e dizer que

$$\mathbf{y} = F\,\mathbf{a},$$

onde F é uma matriz $n \times n$ tal que $F[i, j] = \omega^{ij}$.

* N. de R. Do inglês, *discrete Fourier transform*.

A inversa da transformada discreta de Fourier

A matriz F tem uma inversa F^{-1}, tal que $F^{-1}(F(\mathbf{a})) = \mathbf{a}$ para todo $\mathbf{a}$. A matriz F^{-1} permite a definição da ***inversa da transformada discreta de Fourier***. Se tivermos um vetor $\mathbf{y}$ com os valores de um polinômio p de grau $n - 1$ nas n-ésimas raízes da unidade $\omega^0, \omega^1, \omega^2, ..., \omega^{n-1}$, então podemos recuperar o vetor de coeficientes de p calculando

$$\mathbf{a} = F^{-1}\mathbf{y}.$$

Além disso, a matriz F^{-1} tem uma forma simples, pois $F^{-1}[i, j] = \omega^{-ij}/n$. Assim, podemos recuperar o coeficiente a_i como

$$a_i = \sum_{j=0}^{n-1} y_j \omega^{-ij} / n.$$

O lema a seguir justifica essa afirmação e é o motivo pelo qual chamamos F e F^{-1} de "transformadas".

Lema 10.28: *para qualquer vetor* $\mathbf{a}$, $F^{-1} \cdot F\ \mathbf{a} = \mathbf{a}$.

Prova: seja $A = F^{-1} \cdot F$. É suficiente mostrar que $A[i, j] = 1$ se $i = j$, e $A[i, j] = 0$ se $i \neq j$. Ou seja, $A = I$, onde I é a ***matriz identidade***. Pelas definições de F, F^{-1} e da multiplicação de matrizes,

$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{-ik}\omega^{kj}.$$

Se $i = j$, então essa equação se reduz a

$$A[i,i] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{0} = \frac{1}{n} \cdot n = 1.$$

Assim, considere o caso em que $i \neq j$ e faça $m = j - i$. Então a $i\,j$-ésima entrada da matriz A pode ser escrita como

$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{mk},$$

onde $-n < m < n$ e $m \neq 0$. Pela propriedade do cancelamento para uma n-ésima raiz primitiva da unidade, o lado direito da equação acima se reduz a 0; portanto,

$$A[i, j] = 0$$

para $i \neq j$. ■

Dadas a transformada discreta de Fourier e a inversa da transformada discreta de Fourier, podemos agora definir nossa abordagem para a multiplicação dos polinômios p e q.

O teorema da convolução

A fim de usar a transformada discreta de Fourier e sua inversa para calcular a convolução de dois vetores de coeficientes $\mathbf{a}$ e $\mathbf{b}$, aplicamos os seguintes passos, ilustrados no diagrama da Figura 10.15.

1. Complete **a** e **b** com n zeros e considere-os como vetores-coluna para definir

$$\mathbf{a}' = [a_0, a_1, ..., a_{n-1}, 0, 0, ..., 0]^T$$
$$\mathbf{b}' = [b_0, b_1, ..., b_{n-1}, 0, 0, ..., 0]^T.$$

2. Compute as transformadas discretas de Fourier $\mathbf{y} = F\mathbf{a}'$ e $\mathbf{z} = F\mathbf{b}'$.
3. Multiplique os vetores **y** e **z** componente a componente, definindo o produto $\mathbf{y} \cdot \mathbf{z} = F\mathbf{a}' \cdot F\mathbf{b}'$, onde

$$(\mathbf{y} \cdot \mathbf{z})[i] = (F\mathbf{a}' \cdot F\mathbf{b}')[i] = F\mathbf{a}'[i] \cdot F\mathbf{b}'[i] = y_i \cdot z_i,$$

   para $i = 1, 2, ..., 2n - 1$.
4. Compute a transformada discreta de Fourier inversa do produto anterior, ou seja, compute $\mathbf{c} = F^{-1}(F\mathbf{a}' \cdot F\mathbf{b}')$.

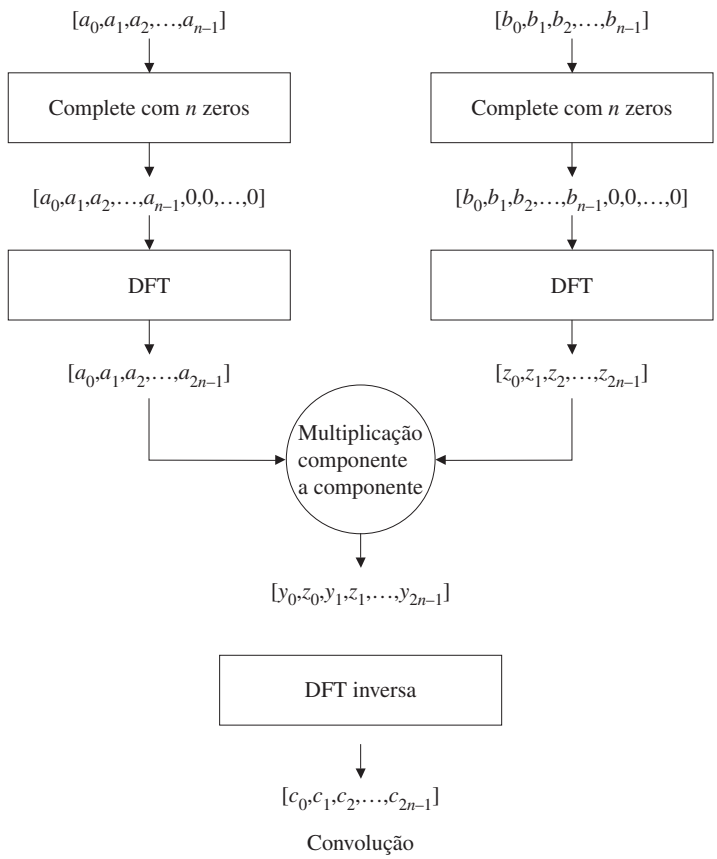


Figura 10.15 Ilustração do Teorema da Convolução para calcular $\mathbf{c} = \mathbf{a} * \mathbf{b}$.

O motivo pelo qual a abordagem anterior funciona é baseado no teorema a seguir.

Teorema 10.29 (Teorema da Convolução): *sejam dois vetores* $\mathbf{a}$ *e* $\mathbf{b}$ *de comprimento n, que, ao serem completados com zeros até o tamanho* $2n$*, originam os vetores* $\mathbf{a}'$ *e* $\mathbf{b}'$*, respectivamente. Então* $\mathbf{a} * \mathbf{b} = F^{-1}(F\mathbf{a}' \cdot F\mathbf{b}')$.

Prova: mostraremos que $F(\mathbf{a} * \mathbf{b}) = F\mathbf{a}' \cdot F\mathbf{b}'$. Considere $A = F\mathbf{a}' \cdot F\mathbf{b}'$. Como as metades finais de $\mathbf{a}'$ e $\mathbf{b}'$ estão preenchidas com zeros,

$$\begin{aligned} A[i] &= \left(\sum_{j=0}^{n-1} a_j \omega^{ij}\right) \cdot \left(\sum_{k=0}^{n-1} b_k \omega^{ik}\right) \\ &= \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)}, \end{aligned}$$

para $i = 0, 1, ..., 2n-1$. Considere em seguida $\mathbf{B} = F(\mathbf{a} * \mathbf{b})$. Pela definição da convolução e pela DFT,

$$B[i] = \sum_{l=0}^{2n-1}\sum_{j=0}^{2n-1} a_j b_{l-j} \omega^{il}.$$

Substituindo k por $l - j$ e alterando a ordem das somas, temos

$$B[i] = \sum_{j=0}^{2n-1}\sum_{k=-j}^{2n-1} a_j b_k \omega^{i(j+k)}.$$

Como b_k é indefinido para $k < 0$, podemos começar a segunda soma acima em $k = 0$. Além disso, como $a_j = 0$ para $j > n - 1$, podemos diminuir o limite superior na primeira soma para $n - 1$. Note que, se fizermos esta substituição, então o limite superior da segunda soma será sempre maior ou igual a n. Então, como $b_k = 0$ para $k > n - 1$, podemos diminuir o limite superior na segunda soma para $n - 1$. Portanto,

$$B[i] = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)},$$

o que prova o teorema. ■

Agora temos um método para computar a multiplicação de dois polinômios que inclui calcular duas DFTs, fazer uma multiplicação componente a componente que leva tempo linear e calcular a DFT inversa. Assim, se acharmos um algoritmo rápido para calcular a DFT e sua inversa, então teremos um algoritmo rápido para a multiplicação de dois polinômios. Descrevemos em seguida um algoritmo desse tipo, que é conhecido como a "transformada rápida de Fourier".

10.4.3 O algoritmo da transformada rápida de Fourier

O algoritmo da ***transformada rápida de Fourier*** (FFT) computa a transformada discreta de Fourier de um vetor de comprimento n em tempo $O(\log(n))$. No algoritmo da FFT, aplicamos a abordagem de divisão e conquista na avaliação de polinômios observando que, se n for par, podemos dividir um polinômio de grau $n - 1$

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

em dois polinômios de grau $n/2 - 1$

$$p^{\text{par}}(x) = a_0 + a_2x + a_4x^2 + ... + a_{n-2}x^{n/2-1}$$
$$p^{\text{ímpar}}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n-1}x^{n/2-1}$$

e notando que podemos combinar esses dois polinômios para obter p, usando a equação

$$p(x) = p^{\text{par}}(x^2) + xp^{\text{ímpar}}(x^2).$$

A DFT avalia $p(x)$ a cada uma das n-ésimas raízes da unidade $\omega^0, \omega^1, \omega^2,..., \omega^{n-1}$. Note que, pela propriedade da redução, os valores $(\omega^2)^0, \omega^2, (\omega^2)^2, (\omega^2)^3,..., (\omega^2)^{n-1}$ são $(n/2)$-ésimas raízes da unidade. Assim, podemos avaliar $p^{\text{par}}(x)$ e $p^{\text{ímpar}}(x)$ nestes valores, e podemos reutilizar esses mesmos cálculos para obter $p(x)$. Essa observação é usada no Algoritmo 10.16 (FFT), que usa como entrada um vetor de coeficientes **a** de tamanho n e uma n-ésima raiz da unidade ω. Para manter a simplicidade, assumimos que n é uma potência de dois.

Algoritmo FFT(**a**, ω):
  Entrada: um vetor de coeficientes $\mathbf{a} = [a_0, a_1,\ldots, a_{n-1}]$ de tamanho n e uma n-ésima raiz primitiva da unidade ω, onde n é uma potência de dois.
  Saída: um vetor **y** de valores do polinômio para **a** nas n-ésimas raízes da unidade.
 se $n = 1$ **então**
   retorne y = a
 $x \leftarrow \omega^0$ { x irá armazenar as potências de ω, então inicialmente $x = 1$. }
 {Etapa de Divisão, que separa índices pares e ímpares }
 $\mathbf{a}^{\text{par}} \leftarrow [a_0, a_2, a_4, ..., a_{n-2}]$
 $\mathbf{a}^{\text{ímpar}} \leftarrow [a_1, a_3, a_5, ..., a_{n-2}]$
 {Chamadas Recursivas, com ω^2 como a $(n/2)$-ésima raiz da unidade, pela propriedade da redução }
 $\mathbf{y}^{\text{par}} \leftarrow$ FFT($\mathbf{a}^{\text{par}}, \omega^2$)
 $\mathbf{y}^{\text{ímpar}} \leftarrow$ FFT($\mathbf{a}^{\text{ímpar}}, \omega^2$)
 {Etapa de Combinação, usando $x = \omega^i$}
 para $i \leftarrow 0$ **a** $n/2 - 1$ **faça**
   $y_i \leftarrow y_i^{\text{par}} + x \cdot y_i^{\text{ímpar}}$
   $y_{i+n/2} \leftarrow y_i^{\text{par}} - x \cdot y_i^{\text{ímpar}}$ {Usa a propriedade reflexiva}
   $x \leftarrow x \cdot \omega$
 retorne y

Algoritmo 10.16 Algoritmo recursivo da FFT.

A correção do algoritmo da FFT

A descrição em pseudocódigo do Algoritmo 10.16 para o algoritmo da FFT é muito simples; portanto, vamos investigar como ele funciona. Primeiramente, note que o caso mais simples da recursão, quando $n = 1$, retorna um vetor **y** correto e com um elemento, $y_0 = a_0$, que é o único termo do polinômio $p(x)$ neste caso.

No caso geral, quando $n \geq 2$, separamos **a** em suas instâncias pares e ímpares, $\mathbf{a}^{\text{par}}$ e $\mathbf{a}^{\text{ímpar}}$, e chamamos FFT recursivamente usando ω^2 como a $(n/2)$-ésima raiz da unidade. Como já mencionamos, a propriedade da redução de uma n-ésima raiz primitiva da unidade permite-nos usar ω^2 desta forma. Assim, podemos assumir indutivamente que

$$y_i^{\text{par}} = p_i^{\text{par}}(\omega^{2i})$$
$$y_i^{\text{ímpar}} = p_i^{\text{ímpar}}(\omega^{2i})$$

Vamos, portanto, considerar o laço **para** que combina os valores das chamadas recursivas. Note que na i-ésima iteração do laço, temos $x = \omega^i$. Assim, quando realizamos a atribuição

$$yi \leftarrow y_i^{\text{par}} + xy_i^{\text{ímpar}},$$

apenas fazemos

$$\begin{aligned} y_i &= p^{\text{par}}((\omega^2)^i) + \omega^i \cdot p^{\text{ímpar}}((\omega^2)^i) \\ &= p^{\text{par}}((\omega^{i+(n/2)})^2) + \omega^{i+n/2} \cdot p^{\text{ímpar}}((\omega^i)^2) \\ &= p(\omega^{i+n/2}), \end{aligned}$$

e fazemos isso para cada índice $i = 0, 1,\ldots, n/2 - 1$. De forma similar, quando realizamos a atribuição

$$y_{i+n/2} \leftarrow y_i^{\text{par}} - xy_i^{\text{ímpar}},$$

apenas fazemos

$$y_{i+n/2} = p^{\text{par}}((\omega^2)^i) - \omega^i \cdot p^{\text{ímpar}}((\omega^2)^i).$$

Como ω^2 é uma $(n/2)$-ésima raiz primitiva da unidade, $(\omega^2)^{n/2} = 1$. Além disso, como ω é por sua vez uma raiz primitiva da unidade,

$$\omega^{i+n/2} = -\omega^i,$$

pela propriedade da reflexão. Assim, podemos reescrever a identidade acima para $y_{i+n/2}$ como

$$\begin{aligned} y_{i+n/2} &= p^{\text{par}}((\omega^2)^{i+(n/2)}) + \omega^i \cdot p^{\text{ímpar}}((\omega^2)^{i+(n/2)}) \\ &= p^{\text{par}}((\omega^{i+(n/2)})^2) + \omega^{i+n/2} \cdot p^{\text{ímpar}}((\omega^{i+(n/2)})^2) \\ &= p(\omega^{i+n/2}), \end{aligned}$$

e isso será válido para cada $i = 0, 1,\ldots, n/2 - 1$. Assim, o vetor **y** retornado pelo algoritmo FFT irá armazenar os valores de $p(x)$ em cada uma das n-ésimas raízes primitivas da unidade.

Análise do algoritmo FFT

O algoritmo FFT segue o paradigma da divisão e conquista, dividindo o problema original de tamanho n em dois problemas de tamanho $n/2$ que são resolvidos recursivamente. Assumimos que cada uma das operações aritméticas realizadas pelos algoritmos leva tempo $O(1)$. A etapa de divisão e a etapa de combinação das soluções recursivas levam tempo $O(n)$; portanto, podemos caracterizar o tempo de execução $T(n)$ do algoritmo FFT usando a recorrência

$$T(n) = 2T(n/2) + bn,$$

para alguma constante $b > 0$. Pelo Teorema Mestre (5.6) $T(n)$ é $O(n \log(n))$. Portanto, podemos resumir a discussão com o teorema a seguir.

Teorema 10.30: *dado um vetor de coeficientes* **a** *de tamanho n definindo um polinômio $p(x)$ e uma n-ésima raiz primitiva da unidade ω, o algoritmo FFT avalia $p(x)$ em cada uma das n-ésimas raízes primitivas da unidade ω^i para $i = 0, 1, \ldots, n - 1$, em tempo $O(n \log(n))$.*

Também existe um algoritmo inverso para FFT, que determina a DFT inversa em tempo $O(n \log(n))$. Os detalhes desse algoritmo são similares aos do algoritmo FFT e são deixados como exercício (R-10.14). Combinando esses dois algoritmos em uma abordagem para multiplicar dois polinômios $p(x)$ e $q(x)$, dados seus vetores de coeficientes com tamanho n, temos um algoritmo para computar o produto em tempo $O(n \log(n))$.

Além disso, a abordagem para usar o algoritmo FFT e sua inversa para determinar o produto de dois polinômios pode ser estendida para o problema da determinação o produto de dois inteiros grandes. Discutiremos essa adaptação a seguir.

10.4.4 Multiplicação de inteiros grandes

Vamos rever o problema estudado na Seção 5.2.2, e supor que temos dois inteiros longos I e J que usam no máximo N *bits* e que estamos interessados em $I \cdot J$. A idéia básica que descrevemos nesta seção é usar o algoritmo de FFT para calcular esse produto. Para usar a FFT desta forma, entretanto, temos de definir a aritmética de inteiros de forma a usar as raízes primitivas da unidade (veja, por exemplo, o Exercício C-10.8).

O algoritmo que descrevemos assume que podemos quebrar I e J em palavras de $O(\log(N))$ *bits*, de tal forma que a aritmética entre as palavras pode ser feita usando-se instruções de máquina de tempo constante. Esta é uma hipótese razoável, pois custa apenas $\lceil \log N \rceil$ *bits* para representar o próprio número N. Assim, faremos a FFT usando aritmética módulo p para um número primo p adequado e que possa ser representado em uma única palavra de tamanho $O(\log(N))$ *bits*. Encontramos o primo p escolhendo pequenos inteiros $c \geq 1$ e $n \geq 1$, tais que $p = cn + 1$ é primo e $N/\lfloor \log p \rfloor \leq n/2$. Para manter a simplicidade da descrição, vamos supor que n é uma potência de dois.

Em geral, esperamos que c seja $O(\log(n))$ na definição de p acima, pois um teorema fundamental na teoria dos números afirma que um número ímpar aleatório na faixa $[1, n]$ é primo com probabilidade $\omega\,(1/\log(n))$. Ou seja, esperamos que p seja representado com $O(\log(N))$ *bits*. Dado esse número primo p, nossa intenção é realizar toda a aritmética da FFT módulo p. Como a realizamos sobre as palavras de um grande vetor de inteiros, também desejamos que o tamanho de cada palavra seja menor do que a metade do que é usado para representar p, pois assim podemos representar o produto de quaisquer duas palavras sem que a aritmética módulo p "apague" um produto de palavras. Por exemplo, os valores de p abaixo são adequados para os computadores atuais:

- Se $n \leq 2^{10} = 1.024$, então escolhemos $p = 25 \cdot 2^{10} + 1 = 25.601$ como o módulo primo, que pode ser representado com uma palavra de 15 *bits*. Além disso, como $p > 2^{14}$ neste caso, essa escolha permite que multipliquemos números cuja representação é tão grande quanto 2^{10} palavras de 7 *bits* cada.
- Se $n \geq 2^{27} = 134.217.728$, então escolhemos $p = 25 \cdot 2^{27} + 1 = 2.013.265.921$ como o módulo primo, que pode ser representado com uma palavra de 31 *bits*. Além disso, como $p > 2^{30}$ neste caso, essa escolha permite que multipliquemos números cuja representação é tão grande quanto 2^{27} palavras de 15 *bits*, ou cerca de 240 Mb.

Dado um primo $p = cn + 1$, para um pequeno valor $c \geq 1$, e n definido de forma que $N/\lfloor \log p \rfloor$ é $O(n)$, definimos $m = \lfloor \log p \rfloor /2$. Interpretamos I e J respectivamente como

vetores **a** e **b** de palavras que usam *m bits* cada, estendidos com quantos zeros forem necessários para fazer com que **a** e **b** tenham comprimento n, com ao menos a metade dos termos superiores do vetor sendo 0. Podemos escrever I e J como

$$I = \sum_{i=0}^{n-1} a_i 2^{mi}$$

$$J = \sum_{i=0}^{n-1} b_i 2^{mi}.$$

Além disso, escolhemos n de forma que os primeiros $n/2$ dos a_i e b_i sejam diferentes de zero. Assim podemos representar o produto $K = I \cdot J$ como

$$K = \sum_{i=0}^{n-1} c_i 2^{mi}.$$

Se tivermos um primo $p = cn + 1$ para um pequeno valor $c \geq 1$, encontramos um inteiro x que é um gerador do grupo Z_p^* (veja a Seção 10.1). Ou seja, encontramos x tal que x^i é diferente para $i = 0, 1, \ldots, p - 1$. Dado um tal gerador x, podemos usar $\omega = x^c \bmod p$ como uma n-ésima raiz primitiva da unidade (assumindo que a multiplicação e a soma são feitas módulo p). Ou seja, cada $(x^c)^i \bmod p$ é diferente para $i = 0, 1, 2, \ldots, n - 1$, mas pelo Pequeno Teorema de Fermat (Teorema 10.8), temos

$$\begin{aligned}(x^c)^n \bmod p &= x^{cn} \bmod p \\ &= x^{p-1} \bmod p \\ &= 1.\end{aligned}$$

Para calcular o produto de dois inteiros I e J de N *bits*, lembre-se ver I e J como dois vetores **a** e **b** de tamanho n com m *bits* cada (com pelo menos os $n/2$ *bits* superiores sendo 0). Note que, já que $m = \lfloor \log p/2 \rfloor$, temos que $2^m < p$. Assim, cada um dos termos em **a** e **b** já está reduzido ao módulo p sem que tenhamos de empregar esforço adicional, e qualquer produto de dois desses termos também está reduzido ao módulo p.

Para computar $K = I \cdot J$, aplicamos o Teorema da Convolução usando os algoritmos da FFT e da FFT inversa para computar a convolução **c** de **a** e **b**. Nesta computação, usamos ω, como definido acima, como a n-ésima raiz primitiva da unidade, e realizamos toda a aritmética (incluindo os produtos das versões transformadas de **a** e **b**) em módulo p.

Os termos na convolução **c** são todos menores do que p, mas para obter um representante de $K = I \cdot J$, precisamos que cada termo seja representado usando exatamente m *bits*. Assim, depois que temos uma convolução **c**, devemos calcular o produto K como

$$K = \sum_{i=0}^{n-1} c_i 2^{mi}$$

Essa computação final não é tão difícil quanto parece, pois multiplicar um número binário por uma potência de dois se reduz a uma operação de *shift*. Além disso, p é $O(2^{m+1})$ e a soma acima envolve apenas a propagação de alguns grupos de *bits* de *carry* de um termo para o seguinte. Assim, a obtenção de uma representação binária de K como um vetor de palavras de m *bits* pode ser feita em tempo $O(n)$. Como a aplicação do Teorema da Convolução descrito acima leva tempo $O(n \log(n))$, temos o resultado a seguir.

Teorema 10.31: *dados dois inteiros I e J de N bits cada, podemos calcular o produto $K = I \cdot J$ em tempo $O(N)$, assumindo que a aritmética envolvendo palavras de tamanho $O(\log(N))$ seja realizada em tempo constante.*

Prova: o número n é escolhido de forma que seja $O(N/\log N)$. Assim, o tempo de execução de $O(n \log(n))$ é $O(N)$, assumindo que a aritmética envolvendo palavras de tamanho $O(\log(N))$ seja realizada em tempo constante. ■

Em alguns casos, não podemos assumir que a aritmética envolvendo palavras de tamanho $O(\log(n))$ seja realizada em tempo constante, mas, em vez disso, temos tempo constante para cada operação entre *bits*. Neste modelo, ainda é possível usar a FFT para multiplicar dois inteiros de N bits, mas os detalhes são mais complicados, e o tempo de execução aumenta para $O(N \log(N) \log \log(N))$.

Na Seção 10.5, estudaremos alguns assuntos importantes com respeito à implementação do algoritmo de FFT, incluindo uma análise experimental de seu comportamento na prática.

10.5 Exemplo em Java: a FFT

Existe um número de tópicos interessantes relacionados ao algoritmo da FFT que discutiremos nesta seção. Começaremos com a descrição de uma classe para inteiros longos que realiza a multiplicação pela versão recursiva da FFT, como descrito no pseudocódigo do Algoritmo 10.16. Mostramos a declaração dessa classe e suas variáveis e constantes no Trecho de Código 10.17. Um detalhe significativo nessas declarações inclui a nossa definição da n-ésima raiz primitiva da unidade, OMEGA. Nossa escolha de 31^{15} vem do fato de que 31 é um gerador para Z_p^* para o número primo $15 \cdot 2^{27}$. Sabemos que 31 é um gerador para esse Z_p^* por causa de um teorema da teoria dos números. Esse teorema diz que um inteiro x é um gerador de Z_p^* se e somente se $x^{\phi(p)/q}$ não for igual a 1 para qualquer divisor primo q de $\phi(p)$, onde $\phi(p)$ é a função totiente de Euler definida na Seção 10.1. Para nosso p em particular, $\phi(p) = 15 \cdot 2^{27}$; portanto, os únicos divisores primos q que devemos considerar são 2, 3 e 5.

```
import java.lang.*;
import java.math.*;
import java.util.*;

public class BigInt {
    protected int signum=0;    //neg = –1, 0 = 0, pos = 1
    protected int[ ] mag;          // magnitude inicia com os digitos menos significativos
    public final static int MAXN=134217728;   // valor máximo para n
    public final static int ENTRYSIZE=10;       // número de bits em cada posição
                                                   de mag
    protected final static long P=2013265921;          // O primo 15*2^{27} + 1
    protected final static int OMEGA=440564289;   // Raiz da unidade 31^{15}
                                                      mod P
    protected final static int TWOINV=1006632961;              //2^{–1} mod P
```

Trecho de Código 10.17 Declarações e variáveis para uma classe de inteiros longos que suporta a multiplicação usando o algoritmo da FFT.

Uma implementação recursiva da FFT

Mostramos o método de multiplicação de nossa classe BigInt para inteiros longos no Trecho de Código 10.18 e, no Trecho de Código 10.19, mostramos a implementação do algoritmo recursivo para FFT. Note que usamos uma variável prod para armazenar o produto que é usado em duas expressões seguintes, e com isso evitamos recalcular a expressão repetida. Além disso, note que toda a aritmética modular é realizada com operações **long**. Essa exigência se deve ao fato de que P usa 31 *bits*, de forma que podemos ter de realizar operações que causariam *overflow* em uma variável inteira. Em vez disso, realizamos toda a aritmética com operações **long** e, ao final, armazenamos o resultado em um **int**.

```
public BigInt multiply(BigInt val) {
    int n = makePowerOfTwo(Math.max(mag.length,val.mag.length)) *2;
    int signResult = signum * val.signum;
    int[ ] A = padWithZeros(mag,n);          // copia mag em A, adicionando zeros
    int[ ] B = padWithZeros(val.mag,n);      // copia val.mag em B, adicionando
                                             zeros
    int[ ] root = rootsOfUnity(n);           // cria as n raizes da unidade
    int[ ] C = new int[n];                   // resultado de A*B
    int[ ] AF = new int[n];                  // resultado para a FFT de A
    int[ ] BF = new int[n];                  // resultado para a FFT de B
    FFT(A, root, n,0,AF);
    FFT(B, root, n,0, BF);
    for (int i=0; i<n; i++)
      AF[i] = (int)(((long)AF[i] *(long) BF[i]) % P);          // multiplicação de
                                                                  componentes
    reverseRoots(root);      // reverte as raízes para criar as raízes inversas
    inverseFFT(AF, root, n, 0, C);      // deixa o resultado da inversa da FFT em C
    propagateCarries(C);   // converte C para o número correto de bits por elemento
    return new Bigint(signResult,C);
    }
```

Trecho de Código 10.18 Método para multiplicação na classe de inteiros longos que suporta a multiplicação pelo algoritmo recursivo da FFT.

```
public static void FFT(int[ ] A, int[ ] root, int n, int base, int[ ] Y) {
    int prod;
    if (n==1) {
      Y[base] = A[base];
      return;
      }
    inverseShuffle(A,n,base);          // desembaralha, para separar pares e ímpares
    FFT(A,root,n/2,base,Y);            // resultados ficam em Y[base] a Y[base+n/2-1]
    FFT(A,root,n/2,base+ n/2,Y);       // resultados ficam em Y[base+n/2] a Y[base+n-1]
    int j = A.length/n;
    for (int i=0; i<n/2; i++) {
      prod = (int) (((long) root [i *j] *Y [base+n /2+i]) % P);
      Y[base+n/2+i] = (int) (((long)Y [base+i] + P - prod) % P);
      Y[base+i] = (int) (((long)Y [base+i] + prod) % P);
```

```
        }
}
public static void inverseFFT(int[ ] A, int[ ] root, int n, int base, int[ ] Y) {
    int inverseN = modInverse(n);                //n^{–1}
    FFT(A,root,n,base,Y);
    for (int i=0; i<n; i++)
      Y[i] = (int)(((long)Y[i]*inverseN) % P);
}
```

Trecho de Código 10.19 Uma implementação recursiva do algoritmo da FFT.

Evitando a alocação repetida de arranjos

O pseudocódigo para o algoritmo recursivo da FFT aloca vários novos arranjos, incluindo $\mathbf{a}^{\text{par}}$, $\mathbf{a}^{\text{ímpar}}$, $\mathbf{y}^{\text{par}}$, $\mathbf{y}^{\text{ímpar}}$ e $\mathbf{y}$. Alocar todos esses arranjos com uma chamada recursiva pode custar um grande trabalho adicional. Se ele puder ser evitado, poderemos melhorar consideravelmente os fatores constantes no tempo $O(n \log(n))$ (ou tempo $O(N)$) do algoritmo da FFT.

Felizmente a estrutura da FFT permite-nos evitar essa alocação repetida de arranjos: em vez de usarmos vários arranjos, usamos o mesmo arranjo A para os coeficientes de entrada e um único arranjo Y para os resultados. A idéia por trás desse uso é podermos pensar em A e Y como se estivessem particionados em subarranjos, cada um associado com uma chamada recursiva diferente. Podemos identificar esses subarranjos usando apenas duas variáveis: base, que identifica o endereço inicial no arranjo, e n, que identifica o tamanho do subarranjo. Assim podemos evitar o custo adicional associado à alocação de muitos arranjos pequenos com cada chamada recursiva.

Desembaralhando

Tendo decidido que não alocaremos novos arranjos durante as chamadas recursivas da FFT, devemos lembrar que esta envolve a realização de computações separadas para as posições pares e ímpares de um arranjo de entrada. No pseudocódigo do Algoritmo 10.16, usamos novos arranjos $\mathbf{a}^{\text{par}}$ e $\mathbf{a}^{\text{ímpar}}$, mas agora devemos usar subarranjos de A como área de armazenamento. Nossa solução para esse problema de gerenciamento de memória é usar o subarranjo atual de n posições em A e dividi-lo em dois subarranjos de tamanho $n/2$. Um dos subarranjos tem a mesma base que A, e o outro tem base base + $n/2$. Movemos os elementos em posições pares de A para a metade inicial e movemos os elementos em posições ímpares de A para a segunda metade. Fazendo isso, definimos uma permutação chamada de ***embaralhamento reverso***. Essa permutação tem esse nome pela semelhança com a inversa da permutação que teríamos se cortássemos o arranjo A no meio e o embaralhássemos (veja a Figura 10.20).

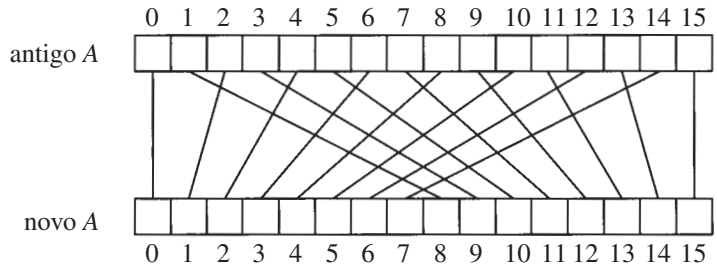


Figura 10.20 Ilustração da permutação do embaralhamento reverso.

Pré-cálculo das raízes da unidade e outras otimizações

Existe uma série de otimizações adicionais que usamos em nossa implementação da FFT. No Trecho de Código 10.21, mostramos alguns dos métodos de suporte importantes para nossa implementação, incluindo a determinação da inversa n^{-1}, a permutação do embaralhamento reverso, o pré-cálculo das n-ésimas raízes primitivas da unidade e a propagação de somas após a convolução ter sido calculada em módulo p.

```
protected static int modInverse(int n) { // assume que n é potência de dois
    int result = 1;
    for (long twoPower = 1; twoPower < n; twoPower *= 2)
      result = (int) (((long)result*TWOINV) % P);
    return result;
}
protected static void inverseShuffle(int[ ] A, int n, int base) {
    int shift;
    int[ ] sp = new int[n];
    for (int i=0; i<n/2; 1++) { // desembaralha A numa área temporária
      shift = base + 2*i;
      sp[i] = A[shift];           // índice par
      sp[i+n/2] = A[shift+1]; // índice ímpar
      }
    for (int i=0; <n; i++)
      A[base+i] = sp[i];        // copia de volta para A
}
protected static int[ ] rootsOfUnity(int n) { // assume que n é potência de dois
    int t = MAXN;
    int nthroot = OMEGA;
    for (int t = MAXN; t>n; t / = 2)     // encontra raiz primitiva da unidade
      nthroot = (int) (((long)nthroot*nthroot) % P);
    int[ ] roots = new int[n];
    int r = 1;  // r vai passar por todas as n-ésimas raízes da unidade
    for (int i=0; i<n; i++) {
      roots[i] = r;
      r = (int) (((long)r*nthroot) % P);
      }
    return roots;
```

```
}
protected static void propagateCarries(int[ ] A) {
    int i, carry;
    carry = 0;
    for (i=0; i<A.length; i++) {
      A[i] = A[i] + carry;
      carry = A[i] >>> ENTRYSIZE;
      A[i] = A[i] – (carry << ENTRYSIZE);
      }
}
```

Trecho de Código 10.21 Métodos de suporte para a FFT recursiva.

Uma implementação iterativa da FFT

Existem reduções de tempo que podem ser feitas no algoritmo da FFT envolvendo a troca da versão recursiva por uma versão iterativa. Nossa FFT iterativa é parte de uma classe alternativa de inteiros longos, chamada de FastInt. O método de multiplicação para essa classe é mostrado no Trecho de Código 10.22.

```
public FastInt multiply(FastInt val) {
    int n = makePowerOfTwo(Math.max(mag.length,val.mag.length)) *2;
    logN = logBaseTwo(n);                    // Log de n na base 2
    reverse = reverseArray(n,logN);          // inicializa tabela reversa de pesquisa
    int signResult = signum * val.signum;
    int[ ] A = padWithZeros(mag,n);          // copia mag em A completando com
                                                zeros
    int[ ] B = padWithZeros(val.mag,n);      // copia mag em B completando com
                                                zeros
    int[ ] root = rootsOfUnity(n);           // cria as n raízes da unidade
    FFT(A,root,n);                           // deixa o resultado da FFT em A
    FFT(B,root,n);                           // deixa o resultado da FFT em B
    for (int i = 0; i<n; i++)
      A[i] = (int) (((long)A[i]*B[i]) % P);  // multiplicação componente a
                                                componente
    reverseRoots(root);                      // reverte as raízes para obter as raizes
                                             inversas
    inverseFFT(A, root, n);  // deixa o resultado da inversa da FFT em A
    propagateCarries(A);    // converte A para o número correto de bits por entrada
    return new Fastint(signResult,A);
}
```

Trecho de Código 10.22 O método da multiplicação para a FFT iterativa.

Cálculo da FFT *in-place*

A partir do Trecho de Código 10.22, podemos perceber uma diferença entre este e o método de multiplicação da versão recursiva da FFT: agora estamos realizando a FFT *in-place*. Ou seja, o arranjo A está sendo usado tanto para a entrada de valores quanto para a saída de resultados. Isso economiza o trabalho adicional de copiar entre os arranjos de entrada e de saída. Além disso, calculamos o logaritmo de n na base 2 e esse

valor fica armazenado em uma variável, pois este logaritmo é usado em repetidas chamadas para o algoritmo iterativo de FFT.

Evitando recursão

O desafio principal para evitar a recursão em uma versão *in-place* do algoritmo da FFT é descobrir uma forma de realizar todos os embaralhamentos no arranjo A de entrada. Em vez de realizar cada embaralhamento reverso com cada iteração, podemos realizar todos os embaralhamentos reversos no início, assumindo que n, o tamanho do arranjo, é uma potência de dois.

Para descobrir como é o efeito total da permutação que teríamos após chamadas recursivas com embaralhamentos reversos sucessivos, vamos considerar a forma como os embaralhamentos reversos movem dados com cada chamada recursiva. Na primeira chamada recursiva, fazemos um embaralhamento reverso em todo o arranjo A. Note como essa permutação opera com os *bits* dos índices de A: todos os elementos que tiverem o último *bit* de seu índice em 0 são colocados na primeira parte de A. Da mesma forma, ele coloca todos os elementos que tiverem o último *bit* de seu índice em 1 na metade superior de A. Ou seja, se um elemento iniciar em uma posição com índice terminando com o *bit* b, então ele termina em uma posição que tem b como seu *bit* mais significativo. No próximo nível de recursão, repetimos o processo em cada metade de A. Analisando os *bits* nesta segunda vez, temos que, para $b = 0, 1$, o embaralhamento recursivo leva elementos originalmente em posições que têm b como seu segundo *bit* menos significativo e os move para posições com b como o segundo *bit* mais significativo. Da mesma forma, na i-ésima vez, o embaralhamento recursivo leva elementos originalmente em posições que têm b como seu i-ésimo *bit* menos significativo e os move para posições com b como o i-ésimo *bit* mais significativo. Assim, se um elemento iniciar na posição com representação binária..., então ele termina na posição que tem a representação binária $[b_{l-1}\ldots b_2 b_1 b_0]$, com $[b_0 b_1 b_2 \ldots b_{l-1}]$, $l = \log_2 n$. Ou seja, podemos realizar todos os embaralhamentos de uma vez apenas movendo elementos em A para a posição cujos *bits* são invertidos em relação à posição inicial. Para realizar essa permutação, montamos um arranjo chamado reverse no método multiply, e o usamos no método bitReversal chamado no método FFT para permutar os elementos no arranjo de entrada A de acordo com a permutação. A versão iterativa resultante do algoritmo da FFT é mostrada no Trecho de Código 10.23.

```
public static void FFT(int[ ] A, int[ ] root, int n) {
    int prod,term, index;    // Valores das subexpressões comuns
    int subSize = 1;         // Tamanho do subproblema
    bitReverse(A,logN);      // Permuta A pela tabela de bits
    for (int lev=1; lev<=logN; lev++) {
      subSize *= 2;          // Dobra o tamanho do subproblema
      for (int base=0; base<n-1; base += subSize) { // Itera os subproblemas
        int j = subSize/2;
        int rootIndex = A.length/subSize;
        for (int i=0; i<j; i++) {
          index = base + i;
          prod = (int) (((long)root[i*rootIndex]*A[index+j]) % P);
          term = A[index];
          A[index+j] = (int) (((long)term + P – prod) % P);
```

```
            A[index] = (int) (((long)term + prod) % P);
            }
          }
        }
}
public static void inverseFFT(int[ ] A, int[ ] root, int n) {
    int inverseN = modInverse(n);                // n^{-1}
    FFT(A,root,n);
    for (int i=0; i<n; i++)
      A[i] = (int) (((long)A[i]*inverseN) % P);
}
```

Trecho de Código 10.23 Implementação iterativa do algoritmo da FFT.

```
protected static void bitReverse(int[ ]  A, int logN) {
    int[ ] temp = new int[A.length];
    for (int i=0; i<A.length; i++)
      temp[reverse[i]] = A[i];
    for (int i=0; i<A.length; i++)
      A[i] = temp[i];
}
protected static int[ ] reverseArray(int n, int logN) {
    int[ ] result = new int[n];
    for (int i=0; i<n; i++)
      result[i] = reverse(i, logN);
    return result;
}
protected static int reverse(int N, int logN) {
    int bit=0;
    int result=0;
    for (int i=0; i<logN; i++) {
      bit = N & 1;
      result = (result << 1) + bit;
      N = N >>> 1;
      }
    return result;
    }
```

Trecho de Código 10.24 Métodos auxiliares para uma implementação iterativa do algoritmo da FFT. Outros métodos auxiliares são iguais aos da implementação recursiva.

Mostramos os método auxiliares usados pelo algoritmo iterativo da FFT no Trecho de Código 10.24. Todos esses métodos auxiliares estão envolvidos na determinação do arranjo reverse que guarda a tabela de permutações e fazem uso dele para realizar a permutação de A.

Resultados experimentais

O objetivo de usar o algoritmo da FFT para multiplicação de inteiros grandes é realizar esta operação em tempo $O(n \log(n))$, em inteiros representados como vetores de n palavras, ao contrário do algoritmo de tempo $O(n^2)$ que é ensinado na escola. Além

disso, desenvolvemos uma versão iterativa do algoritmo da FFT para melhorar os fatores constantes no tempo de execução desse método de multiplicação. Para testar esses objetivos na prática, vamos realizar um experimento simples.

Neste experimento, geramos aleatoriamente 10 inteiros grandes consistindo em 2^s palavras de 15 *bits* cada para $s = 7, 8,..., 16$, e os multiplicamos em pares consecutivos, produzindo nove produtos para cada valor de s. Medimos o tempo de execução da avaliação desses produtos e comparamos esses tempos com o tempo para efetuar as multiplicações com a classe BigInteger de Java. Os resultados do experimento são mostrados na Figura 10.25. Os tempos de execução são mostrados em milissegundos. O experimento foi realizado em uma máquina virtual Sun Java com JDK 1.2 em uma Sun Ultra5 com um processador de 360 MHz e 128 MB de memória.

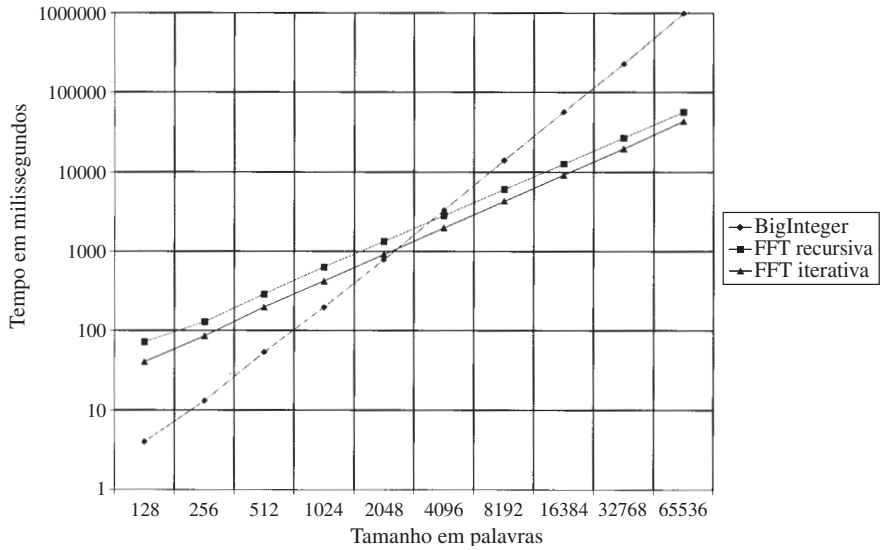


Figura 10.25 Tempos de execução para a multiplicação de inteiros. A tabela mostra os tempos de execução para a multiplicação de inteiros formados de vários números com palavras de 15 *bits* (para a FFT) ou números de tamanho equivalente para uma implementação de BigInteger. Note que as escalas são logarítmicas nos eixos x e y.

Note que mostramos os resultados do experimento em uma escala log-log. Esta escolha corresponde ao teste da potência (Seção 1.6.2) e permite que verifiquemos os custos relativos das diferentes implementações mais claramente do que em uma escala linear. Como $\log n^c = c \log n$, a inclinação de uma reta na escala log-log corresponde exatamente ao grau de um polinômio em uma escala linear, e a altura de uma reta nesta escala corresponde a uma constante de proporcionalidade. A tabela da Figura 10.25 mostra os tempos usando base 10 no eixo y e base 2 no eixo x. Usando o fato de que $\log_2 10$ é aproximadamente 3.322, notamos que o tempo de execução do algoritmo usual de multiplicação é $\Theta(n^2)$, enquanto o tempo do algoritmo com FFT é quase linear. Também notamos que os fatores constantes na implementação da FFT iterativa são cerca de 70% dos fatores constantes na implementação recursiva do algoritmo da FFT. Note a diferença entre os métodos baseados em FFT e o algoritmo usual de multiplicação: para problemas pequenos, os métodos baseados em FFT são 10 vezes mais lentos que o algoritmo usual, e para problemas grandes, eles são mais de 10 vezes mais rápidos.

10.6 Exercícios

Reforço

R-10.1 Prove o Teorema 10.1.

R-10.2 Mostre a execução do método EuclidGCD(14300, 5915) construindo uma tabela similar à Tabela 10.2.

R-10.3 Escreva uma versão não-recursiva do algoritmo EuclidGCD.

R-10.4 Mostre a execução do método EuclidBinaryGCD(14300, 5915) construindo uma tabela similar à Tabela 10.2.

R-10.5 Mostre a existência de inversos aditivos em Z_p, ou seja, prove que, para cada $x \in Z_p$, existe um $y \in Z_p$ tal que $x + y \bmod p = 0$.

R-10.6 Construa a tabela de multiplicação dos elementos de Z_{11}, onde o elemento na linha i e coluna j ($0 \leq i, j \leq 10$) é dado por $i \cdot j \bmod 11$.

R-10.7 Prove o Corolário 10.7.

R-10.8 Forneça provas alternativas do Teorema 10.6 e do Corolário 10.7 que não usem o Teorema 10.3.

R-10.9 Mostre a execução do método FastExponentiation(5, 12, 13) construindo uma tabela similar à Tabela 10.8.

R-10.10 Escreva uma versão não-recursiva do algoritmo ExtendedEuclidGCD.

R-10.11 Aumente a Tabela 10.10 com duas linhas que contenham os valores de ia e jb a cada passo do algoritmo e verifique que $ia + jb = 1$.

R-10.12 Mostre a execução do método ExtendedEuclidGCD(412, 113) construindo uma tabela similar à Tabela 10.10.

R-10.13 Determine os inversos multiplicativos dos números 113, 114 e 127 em Z_{299}.

R-10.14 Descreva o algoritmo inverso da FFT, que calcula a DFT inversa em tempo $O(n \log(n))$. Ou seja, mostre como inverter os papéis de **a** e **y** e mude as atribuições de forma que, para cada índice i, tenhamos

$$a_i = \frac{1}{n}\sum_{j=1}^{n-1} y_j \omega^{-ij}.$$

R-10.15 Prove uma forma mais geral da propriedade de redução das raízes primitivas da unidade, mostrando que para cada inteiro $c > 0$, se ω é uma cn-ésima raiz da unidade, então ω^c é uma n-ésima raiz primitiva da unidade.

R-10.16 Escreva sob a forma $a + b\mathbf{i}$ as n-ésimas raizes complexas da unidade para $n = 4$ e $n = 8$.

R-10.17 Qual a permutação que reverte os *bits* (reverse) para $n = 16$?

R-10.18 Use a FFT e a FFT inversa para calcular a convolução de **a** = [1, 2, 3, 4] e **b** = [4, 3, 2, 1]. Mostre o resultado de cada componente como na Figura 10.15.

R-10.19 Use o teorema da convolução para calcular o produto dos polinômios $p(x) = 3x^2 + 4x + 2$ e $q(x) = 2x^3 + 3x^2 + 5x + 3$.

R-10.20 Calcule a transformada discreta de Fourier do vetor [5, 4, 3, 2] usando aritmética módulo $17 = 2^4 + 1$. Use o fato de que 5 é um gerador de Z_{17}^*.

R-10.21 Construa uma tabela mostrando um exemplo do criptossistema RSA com parâmetros $p = 17$, $q = 19$ e $e = 5$. A tabela deve ter duas linhas, uma para o texto M e outra para o texto cifrado C. As colunas devem corresponder a valores inteiros na faixa [10, 20] para M.

Criatividade

C-10.1 Justifique o funcionamento do algoritmo binário de Euclides (Algoritmo 10.3) e analise seu tempo de execução.

C-10.2 Seja p um primo ímpar.

a. Mostre que Z_p tem exatamente $(p - 1)/2$ resíduos quadráticos.

b. Mostre que

$$\left(\frac{a}{b}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}.$$

Forneça um algoritmo randomizado para encontrar um resíduo quadrático de Z_p em tempo esperado $O(1)$.

Discuta a relação entre resíduos quadráticos e a técnica de teste quadrático usada para resolução de colisões em tabelas de *hash* (veja a Seção 2.5.5 e o Exercício C-2.35).

C.10.3 Seja p um número primo. Forneça um algoritmo alternativo eficiente para calcular o inverso multiplicativo de um elemento de Z_p que não seja baseado no algoritmo estendido de Euclides. Qual o tempo de execução de seu algoritmo?

C-10.4 Mostre como modificar o algoritmo ExtendedEuclidGCD para calcular o inverso multiplicativo de um elemento em Z_n usando operações aritméticas em operandos com no máximo $2\lceil \log_2 n \rceil$ *bits*.

C-10.5 Demonstre o funcionamento correto do algoritmo Jacobi (a, b) (Algoritmo 10.12) para calcular o símbolo de Jacobi. Também demonstre que o algoritmo executa $O(\log \max(a, b))$ operações aritméticas.

C-10.6 Forneça uma descrição em pseudocódigo da função de testemunho de composição do algoritmo de Rabin-Miller.

C-10.7 Descreva um algoritmo de divisão e conquista que não seja baseado na FFT para multiplicação de dois polinômios de grau n com coeficientes inteiros em tempo $O(n^{\log_2 3})$ assumindo que as operações aritméticas entre inteiros têm tempo constante.

C-10.8 Prove que $\omega = 2^{4b/m}$ é uma m-ésima raiz primitiva da unidade quando a multiplicação é feita em módulo $(2^{2b} + 1)$ para qualquer inteiro $b > 0$.

C-10.9 Dados os polinômios $p(x)$ e $q(x)$ de grau n, descreva um algoritmo de tempo $O(n \log(n))$ para multiplicar a derivada de $p(x)$ pela derivada de $q(x)$.

C-10.10 Descreva uma versão da FFT que funcione quando n é uma potência de 3 dividindo o vetor de entrada em três subvetores, fazendo a recursão em cada um e unindo as soluções dos subproblemas. Obtenha uma equação de recorrência para o tempo de execução deste algoritmo e a resolva usando o Teorema Mestre.

C-10.11 Suponha que temos um conjunto de números reais $X = \{x_0, x_1, ..., x_{n-1}\}$. Note que, pelo Teorema da Interpolação Polinomial, existe um único polinômio $p(x)$ de grau $n - 1$ tal que $p(x_i) = 0$ para $i = 0, 1, ..., n - 1$. Descreva um algoritmo de divisão e conquista que obtenha uma representação por coeficientes para $p(x)$ em tempo $O(n \log^2 n)$.

Projetos

P-10.1 Escreva uma classe que contém os métodos para exponenciação modular e cálculo de inversos multiplicativos.

P-10.2 Implemente o algoritmo randomizado para teste de primalidade de Rabin-Miller e de Solovay-Strassen. Teste os resultados desses algoritmos em inteiros de 32 *bits* gerados aleatoriamente usando parâmetros de confiança 7, 10 e 20, respectivamente.

P-10.3 Implemente uma versão simplificada do sistema RSA para mensagens inteiras com uma classe Java que forneça métodos de encriptação, decriptação, assinatura e verificação de assinaturas.

P-10.4 Implemente uma versão simplificada do sistema El Gamal para mensagens inteiras com uma classe Java que forneça métodos de encriptação, decriptação, assinatura e verificação de assinaturas.

Notas

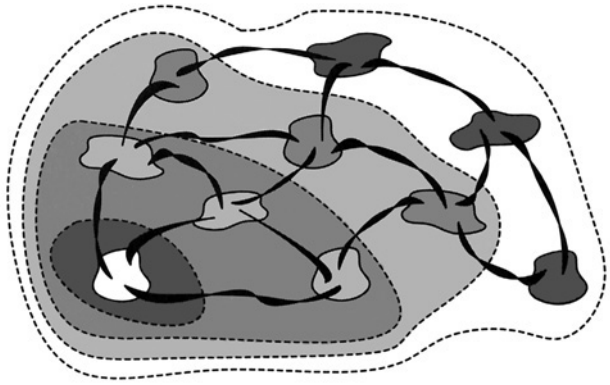
Uma introdução à teoria dos números é dada nos livros de Koblitz [123] e de Kranakis [125]. O livro clássico sobre algoritmos numéricos é o segundo volume da série de Knuth ***The Art of Computer Programming*** [121]. Algoritmos para problemas da teoria dos números também são apresentados nos livros de Brassoud e Wagon [40] e de Bach e Shallit [20]. O algoritmo randomizado de Solovay-Strassen para teste de primalidade aparece em [190, 191]. O algoritmo de Rabin-Miller aparece em [171].

O livro de Schneier [180] descreve os algoritmos e protocolos criptográficos em detalhes. Aplicações da criptografia para segurança em redes são descritas no livro de Stallings [192]. O sistema RSA recebeu esse nome das iniciais de seus inventores, Rivest, Shamir e Adleman [173]. O sistema El Gamal recebeu o nome de seu inventor [75]. A estrutura da árvore *hash* foi introduzida por Merkle [153, 154]. A função ***acumuladora*** de sentido único é introduzida em [27, 179].

A transformada rápida de Fourier (FFT) aparece em um artigo de Cooley and Tukey [54]. Ela também é discutida nos livros de Aho, Hopcroft e Ullman [7], Baase [19], Cormen, Leiserson e Rivest [55], Sedgewick [182, 183] e Yap [213], todos influentes na discussão acima. Em especial, a implementação do algoritmo para multiplicação rápida de inteiros recebeu influência do algoritmo QuickMul de Yap e Li. Para informação adicional sobre outros usos da FFT, o leitor pode consultar os livros de Brigham [41] e Elliot e Rao [66], e o capítulo de Emiris e Pan [67]. As Seções 10.1 e 10.2 foram baseadas parcialmente em uma seção de um artigo não publicado de Achter e Tamassia [1].

Capítulo

11 Algoritmos de redes



11.1 Modelos e medidas de complexidade 506

11.1.1 As camadas do protocolo de rede 506

11.1.2 O modelo de troca de mensagens 507

11.1.3 Medidas de complexidade para algoritmos de redes 509

11.2 Algoritmos distribuídos fundamentais 510

11.2.1 Eleição do líder em um anel 510

11.2.2 Eleição do líder em uma árvore 513

11.2.3 Pesquisa em largura 516

11.2.4 Árvores de cobertura mínima 520

11.3 Roteamento *broadcast* e *unicast* 522

11.3.1 Algoritmo da inundação para roteamento *broadcast* 522

11.3.2 Algoritmo do vetor de distâncias para roteamento *unicast* 524

11.3.3 Algoritmo *link-state* para roteamento *unicast* 525

11.4 Roteamento *multicast* 526

11.4.1 Caminhamento reverso 527

11.4.2 Árvores centradas 528

11.4.3 Árvores de Steiner 529

11.5 Exercícios 532

Algoritmos de redes são escritos para executar em redes de computadores, incluindo aqueles que especificam a forma como computadores especializados, chamados de ***roteadores***, enviam pacotes de dados pela rede. Uma rede de computadores é modelada como um grafo, com os vértices representando processadores ou roteadores e as arestas representando canais de comunicação (veja a Figura 11.1). Algoritmos que executam em tais redes não são convencionais, porque suas entradas estão normalmente espalhadas pela rede, e sua execução é distribuída entre os processadores da rede.

Desta forma, começaremos nosso estudo sobre algoritmos de rede considerando alguns algoritmos distribuídos fundamentais. Estes fazem referência a alguns conceitos básicos encontrados em algoritmos de rede, incluindo a eleição do líder e a construção de árvores de cobertura. Examinaremos cada algoritmo tanto em uma configuração síncrona, em que os processadores operam por "passos" em séries de "rodadas", como em configurações assíncronas, em que os processadores podem executar em velocidades relativas diferentes. Em alguns casos, lidar com a falta de sincronismo insere complicações, mas em redes heterogêneas, tais como a Internet, a falta de sincronismo é uma realidade que deve ser tratada de forma efetiva.

A principal categoria dos algoritmos de redes são os algoritmos de ***roteamento***, que especificam como mover pacotes de informação entre os vários computadores de uma rede. Bons algoritmos de roteamento podem orientar os pacotes de maneira que os mesmos cheguem a seus destinos de forma rápida e confiável, ao mesmo tempo sendo "justos" com os demais pacotes da rede. Um desafio para o projeto de bons algoritmos de roteamento é que esses objetivos são um tanto quanto contraditórios, uma vez que a justiça, algumas vezes, conflita com a velocidade na entrega dos pacotes. Outro desafio é que o processamento deve implementar um esquema de roteamento que deve ser rápido e eficiente. Por essa razão, focamos a segunda parte deste capítulo em algoritmos de roteamento, incluindo métodos para os seguintes padrões de comunicação:

- ***roteamento broadcast:*** envia pacotes para todos os computadores;
- ***roteamento unicast:*** envia pacotes para um computador específico;
- ***roteamento multicast:*** envia pacotes para um grupo de computadores.

Estudaremos esses algoritmos de roteamento analisando os custos envolvidos em sua configuração e no roteamento das mensagens. Antes de discutirmos como esses e outros algoritmos de rede trabalham, entretanto, devemos explicar um pouco mais sobre como opera o modelo computacional no qual são projetados.

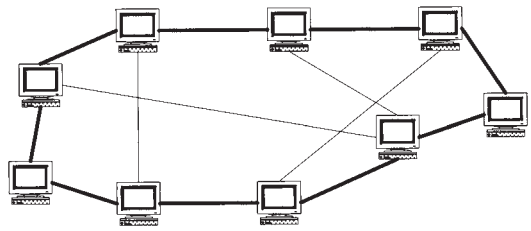


Figura 11.1 Uma rede computacional cujos vértices são processadores e as arestas são conexões de comunicação. As arestas em negrito definem um ciclo simples conhecido como rede em anel.

11.1 Modelos e medidas de complexidade

Antes de podermos estudar algoritmos de redes a fundo, necessitamos entender melhor como as redes, tais como a Internet, funcionam. Mais especificamente, necessitamos explorar como a comunicação entre processadores é executada, modelada e analisada.

11.1.1 As camadas do protocolo de rede

Uma forma de entender como uma rede tal como a Internet funciona é utilizando um modelo de camadas com os seguintes níveis conceituais:

- ***Nível físico:*** este é o nível em que os *bits* são transmitidos por um meio físico, tal como fios elétricos ou cabos de fibra ótica. As decisões de projeto neste nível têm mais a ver com as técnicas de engenharia para representar zeros e uns, maximizando a largura de transmissão e minimizando o ruído.
- ***Nível de enlace:*** este é o nível que lida com os métodos que quebram os dados em subunidades, chamadas de ***frames***, e transmitem dados de um computador, por nível físico, para outro. Um exemplo típico inclui o protocolo ponto a ponto que descreve como enviar dados entre dois computadores conectados fisicamente.
- ***Nível de rede:*** este nível está relacionado com os métodos que quebram os dados em ***pacotes*** (diferentes de *frames*) e conduzem esses pacotes de um computador para outro pela rede. As questões algorítmicas relativas a este nível estão relacionadas a métodos de roteamento de pacotes pela rede e com a quebra e a remontagem de pacotes. O principal protocolo usado neste nível é o ***Internet Protocol*** ou ***IP***. Esse protocolo usa a abordagem do "esforço máximo", o que significa que não são executadas garantias de desempenho e que alguns pacotes eventualmente podem ser perdidos.
- ***Nível de transporte:*** este nível aceita dados de várias aplicações, divide-os em unidades menores se necessário e repassa-os para o nível de rede de uma forma que garanta o controle de confiabilidade e congestionamento. Os dois principais protocolos que existem neste nível são o ***Transmission Control Protocol*** (***TCP***), que provê uma transmissão na Internet orientada por conexão livre de erros e fim a fim entre duas máquinas, e o ***User Datagram Protocol*** (***UDP***), que é um protocolo de esforço máximo não orientado por conexão.
- ***Nível de aplicação:*** este é o nível em que as aplicações operam, usando protocolos de baixo nível. Exemplos de aplicações para Internet incluem *e-mail* (SMTP), transferência de arquivos (FTP e SCP), terminais virtuais (TELNET e SSH), a rede mundial de computadores (HTTP) e o serviço de nomes de domínios (DNS), que mapeia nomes de hospedeiros (tais como www.cs.brown.edu) para endereços IP (tais como 128.148.32.110).

11.1.2 O modelo de troca de mensagens

Falaremos muito pouco a respeito das camadas do protocolo de rede neste capítulo. Em vez disso, os algoritmos de rede que discutiremos operam em um modelo computacional que abstrai as funcionalidades fornecidas pelas várias ***camadas do protocolo de rede*** apresentadas anteriormente. Entretanto, o entendimento das diferentes camadas do protocolo de rede nos permite unir conceitos em nosso modelo computacional distribuído para especificar as funções e os protocolos sobre uma rede real, tal como a Internet.

Existem diferentes formas de abstrair as funcionalidades de uma rede de forma que nos permita descrever algoritmos de redes. Neste capítulo, usaremos o ***modelo de troca de mensagens*** que é normalmente utilizado na descrição de ***algoritmos distribuídos***, em que processadores fracamente acoplados se comunicam visando à solução de problemas computacionais. O modelo alternativo de ***algoritmos paralelos*** é apresentado na Seção 14.2. Em um algoritmo paralelo, processadores fortemente acoplados cooperam e se coordenam normalmente usando um espaço compartilhado de memória.

No modelo de troca de mensagens, que será nosso foco exclusivo neste capítulo, uma rede é modelada como um grafo, onde os vértices correspondem a processadores e as arestas correspondem a conexões fixas entre processadores que não se alteram durante o processamento. Cada aresta e suporta a troca de mensagens entre os dois processadores associados às extremidades de e. As mensagens podem ser enviadas apenas em uma direção, se e for orientada, e nas duas direções, se e não for orientada. Por exemplo, no nível de rede da Internet, os vértices correspondem a computadores especializados chamados de ***roteadores IP***, cada um identificado por um código numérico único chamado de ***endereço IP***, e as mensagens trocadas entre os mesmos são as unidades básicas de transmissão de dados conhecidas como ***pacotes IP***. Outra possibilidade são os vértices corresponderem a servidores na Internet e as arestas a conexões IP permanentes que são mantidas entre vários pares de servidores.

Como o próprio nome diz, no modelo de troca de mensagens, a informação trafega pela rede pela troca de mensagens. Cada processador na rede é associado a um ***identificador*** numérico único (por exemplo, pode ser o endereço IP de um roteador ou de um hospedeiro na Internet). Além disso, assumimos que cada processador conhece seus vizinhos na rede, e que se comunica diretamente apenas com esses vizinhos. Em alguns casos, podemos permitir que cada processador conheça o número total de processadores na rede, mas na maioria das vezes esse conhecimento global não está disponível ou é desnecessário.

Outra consideração importante em algoritmos de redes é se um algoritmo deve ou não lutar contra potenciais mudanças na rede durante sua execução. Durante a execução de um algoritmo, processadores podem falhar (ou "pendurar"), conexões de rede podem cair, e os processadores podem permanecer ativos mas executando computações falhas ou mal-intencionadas. No caso ideal, gostaríamos que os algoritmos distribuídos fossem dinâmicos, ou seja, capazes de responder a alterações na rede e tolerar falhas, isto é, capazes de se recuperar corretamente e elegantemente das falhas. Entretanto, as técnicas para construir algoritmos distribuídos realmente robustos são bastante complexas; logo, estão fora do escopo deste livro. Assim, neste capítulo assumiremos que a topologia da rede não se altera e que os processadores permanecem ativos e

sem falhas enquanto a rede estiver executando um algoritmo distribuído. Isto é, assumimos que a rede permanece ***estática*** durante a execução do algoritmo.

Uma das considerações mais importantes no modelo de troca de mensagens para computação distribuída é a premissa que fazemos sobre a sincronização dos processadores da rede. Existem diferentes possibilidades em relação à sincronização, mas as que seguem são os modelos mais utilizados nos projetos de algoritmos distribuídos:

- ***Modelo síncrono:*** neste modelo, cada processador tem um relógio interno que controla a execução do programa, sendo que os relógios de todos os processadores estão sincronizados. Além disso, assumimos que as velocidades de execução dos processadores são uniformes e que cada processador leva a mesma quantidade de tempo para executar a mesma operação (tal como adição ou comparação). Finalmente, assumimos que leva o mesmo tempo enviar uma mensagem por qualquer conexão da rede.
- ***Modelo assíncrono:*** neste modelo, não inferimos nada em relação aos relógios internos que os processadores possam ter. Além disso, não assumimos que as velocidades internas dos processadores sejam necessariamente similares. Assim, os passos de um algoritmo assíncrono são determinados por condições ou ***eventos***, não pelos pulsos do relógio. Por outro lado, existem algumas premissas razoáveis em relação ao tempo que fazemos de maneira a permitir que os algoritmos executem suas tarefas de forma efetiva. Primeiramente assumimos que cada canal de comunicação, representado por uma aresta, é uma fila do tipo primeiro-que-entra–primeiro-que-sai (FIFO – *first-in*, *first-out*) que pode armazenar em um *buffer* um número não determinado de mensagens. Isto é, as mensagens enviadas por uma aresta são armazenadas na mesma ordem em que foram enviadas. Em segundo lugar, assumimos que, apesar da velocidade dos processadores poder variar, eles não são arbitrariamente diferentes um dos outros. Isto é, existe uma premissa ***fraca*** que garante que, se uma processador p tem um evento que o habilita a executar uma tarefa, então p fatalmente irá executar esta tarefa.

Além dessas duas versões extremas do modelo de computação distribuída, podemos considerar também versões intermediárias, em que os processadores têm relógios – mas esses relógios não estão perfeitamente sincronizados. Assim, nesses modelos intermediários, não podemos assumir que os processadores trabalham em passos sincronizados, mas podemos, contudo, fazer escolhas baseados em eventos de "estouro de tempo" em que os processadores não respondem após um determinado período de tempo. Apesar desses modelos intermediários corresponderem mais fielmente às propriedades de tempo das redes reais, tais como a Internet, achamos mais útil, entretanto, nos restringir aos modelos síncrono e assíncrono descritos anteriormente. Algoritmos projetados para o modelo síncrono são normalmente simples e, em geral, podem ser convertidos em algoritmos que trabalham com o modelo assíncrono. Além disso, algoritmos projetados para o modelo assíncrono certamente funcionarão em modelos intermediários que têm sincronização limitada.

11.1.3 Medidas de complexidade para algoritmos de redes

No projeto tradicional de algoritmos, como estudado em outros capítulos deste livro, as principais medidas de complexidade usadas para determinar a eficiência de um algoritmo são o tempo de execução e a quantidade de memória utilizada. Essas medidas de complexidade não se traduzem diretamente para o domínio dos algoritmos de redes, uma vez que eles assumem implicitamente que os cálculos são feitos em um único computador, e não em uma rede. Em algoritmos de redes, as entradas estão espalhadas pelos computadores da rede e os cálculos podem ser transportados por eles. Assim, devemos tomar alguns cuidados para caracterizar os parâmetros de desempenho de um algoritmo de rede.

As medidas de complexidade para avaliar algoritmos tradicionais têm equivalentes em algoritmos distribuídos, mas são medidas complexas e únicas para uma determinada configuração de rede. Neste capítulos, focamos as seguintes medidas de complexidade:

- ***Ciclos computacionais***: vários algoritmos de rede executam por uma série de ciclos globais até finalmente convergir para uma solução. O número de ciclos necessários para a convergência pode ser, por esta razão, usado como uma aproximação grosseira do tempo de execução. Em algoritmos síncronos, esses ciclos são determinados por pulsos de relógio, enquanto que em algoritmos assíncronos, esses ciclos são quase sempre determinados pela propagação de "ondas" de eventos pela rede.
- ***Espaço***: a quantidade de espaço necessário para um cálculo pode ser usada mesmo em algoritmos de rede. Ela, entretanto, necessita ser qualificada para especificar se é o limite ***global*** do espaço total usado por todos os computadores envolvidos no algoritmo ou um limite ***local*** referente à quantidade de espaço necessário em cada computador envolvido.
- ***Tempo local de execução***: embora seja difícil analisar o tempo total de execução necessário para um cálculo, em especial no caso de algoritmos assíncronos, podemos, entretanto, analisar a quantidade de tempo local necessário para um computador em particular participar de um algoritmo de rede. Se todos os computadores da rede estiverem executando basicamente o mesmo tipo de função, então um limite de tempo de execução local único será suficiente para todos. Mas se por outro lado existirem várias classes de computadores participando do algoritmo, deveremos caracterizar o tempo local necessário para cada classe de computador.
- ***Complexidade das mensagens***: este parâmetro mede o número total de mensagens (cujo tamanho é no máximo logarítmico em relação ao total de processadores) que são enviadas entre pares de computadores durante uma computação. Por exemplo, se uma mensagem M é roteada através de p arestas para ir de um computador a outro, podemos dizer que a complexidade da mensagem para esta comunicação é $p|M|$, onde $|M|$ denota o tamanho de M (em palavras).

Finalmente, se existe alguma outra medida de complexidade óbvia que estivermos tentando otimizar em um algoritmo de rede (tal como lucros em um protocolo de leilão *on-line* ou custo de recuperação em um esquema de conta corrente *on-line*), então poderemos incluir limites para essas medidas da mesma forma.

11.2 Algoritmos distribuídos fundamentais

Medidas de complexidade para algoritmos de rede são em geral mais bem compreendidas como funções de alguma noção intuitiva do "tamanho" do problema que estamos tentando solucionar. Por exemplo, o tamanho de um problema pode ser expresso em função dos seguintes parâmetros:

- número de palavras usado para expressar a entrada;
- número de processadores alocados;
- quantidade de comunicações entre os processadores.

Assim, para analisar corretamente as medidas de complexidade de um algoritmo de rede, devemos formalizar o que entendemos por tamanho do problema para um dado algoritmo.

Para tornar essas noções mais concretas, estudaremos vários algoritmos distribuídos fundamentais nesta seção, analisando *seu desempenho* e usando algumas das medidas de complexidade listadas. Os problemas específicos que estudaremos são escolhidos para demonstrar tanto técnicas fundamentais de distribuição de algoritmos como métodos para analisá-los.

11.2.1 Eleição do líder em um anel

O primeiro problema que iremos abordar é a ***eleição de um líder*** em uma rede em anel. Neste problema, temos uma rede de n processadores conectados em um anel, isto é, o grafo da rede é um ciclo simples com n vértices. O objetivo é identificar um dos n processadores como o "líder", e fazer com que todos os outros processadores concordem com essa seleção. Descreveremos o algoritmo para o caso em que o anel é um ciclo orientado. O Exercício C-11.1 considera o caso de um anel não-orientado.

Começamos pela descrição de uma solução síncrona. A idéia principal da solução síncrona é selecionar como líder o processador que tiver o menor identificador. O desafio é que, em um anel, não existe um ponto óbvio para começar. Então, iniciamos a computação em qualquer lugar. No início do algoritmo, cada computador envia seu identificador para o próximo processador do anel. Nos ciclos subseqüentes, cada processador executa as seguintes computações:

1. recebe um identificador i de seu predecessor no anel;
2. compara i com seu próprio identificador;
3. envia o menor destes dois valores a seu sucessor no anel.

Se um processador receber seu próprio identificador de seu predecessor, então este processador sabe que tem o menor identificador e, portanto, ele é o líder. Esse processador pode então enviar uma mensagem pelo anel informando a todos que ele é o líder.

Fornecemos uma descrição deste método no Algoritmo 11.2. Um exemplo da execução do algoritmo é ilustrado na Figura 11.3.

Algoritmo RingLeader(id):
  Entrada: o identificador único, id, do processador que está executando o algoritmo.
  Saída: o menor identificador de um processador na rede.
  $M \leftarrow$ [Candidato é id]
  Envia a mensagem M ao processador sucessor na rede.
  $feito \leftarrow$ **falso**
  repita
    Pegue a mensagem M oriunda do processador predecessor na rede.
    se M = [Candidato é i] **então**
      se $i = id$ **então**
        $M \leftarrow$ [Líder é id]
        $feito \leftarrow$ **verdadeiro**
        senão
          $m \leftarrow \min\{i,id\}$
          $M \leftarrow$ [Candidato é m]
    senão
      {M é uma mensagem "O líder é"}
      $feito \leftarrow$ **verdadeiro**
    Envia mensagem M para o próximo processador do anel.
    até $feito$
    retorne M {M é uma mensagem "O líder é"}

Algoritmo 11.2 Um algoritmo síncrono executado por cada um dos processadores em um anel orientado para determinar o "líder" de um anel de processadores.

Antes de analisarmos o algoritmo RingLeader, vamos primeiramente nos convencer de que ele funciona corretamente. Seja n o número de processadores. Cada processador inicia enviando seu identificador no primeiro ciclo. Então, nos próximos $n - 1$ ciclos, cada processador aceita uma mensagem "é candidato", calcula o mínimo m entre o identificador i da mensagem e seu próprio identificador id, e transmite m para o próximo processador usando uma mensagem "é candidato". Seja ℓ o menor identificador de um processador no anel. A primeira mensagem enviada pelo processador ℓ irá percorrer todo o anel e voltar inalterada para o processador ℓ. Neste ponto, o processador ℓ irá perceber que é o líder e irá informar todos os outros processadores usando uma mensagem "o líder é" que irá percorrer o anel nos próximos n ciclos. Deve-se observar que o número n de processadores usado nesta análise não precisa ser conhecido pelos processadores. Além disso, observe que não existem *deadlocks* no algoritmo, isto é, não existem dois processadores que estejam ambos aguardando por mensagens que devem ser enviadas pelo outro.

Vamos analisar agora o desempenho do algoritmo RingLeader. Primeiramente, em função do número de ciclos, a primeira mensagem "é candidato" que parte do líder leva n ciclos para percorrer o anel. Além disso, a mensagem "o líder é" enviada pelo líder leva outros n ciclos para percorrer todos os demais processadores. Assim, existem $2n$ ciclos.

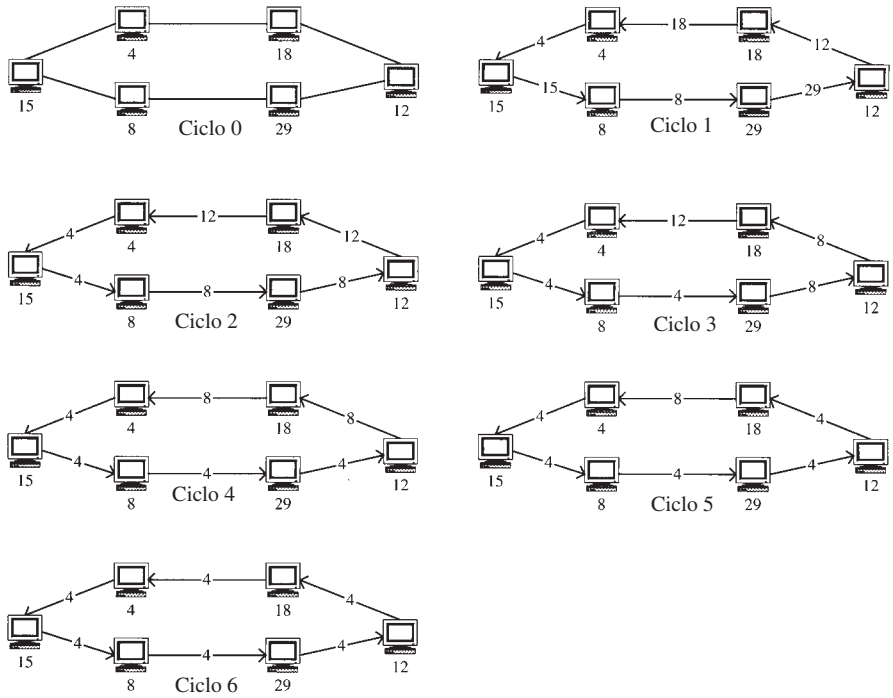


Figura 11.3 Ilustração do algoritmo para eleição síncrona do líder em um anel. A configuração inicia é rotulada de "Ciclo 0". Para cada ciclo sucessivo, rotulamos cada aresta com uma mensagem "é candidato" enviada pela aresta. Não mostramos os ciclos finais, em que o processador 4 informa aos demais que ele é o líder usando uma mensagem "o líder é".

Analisemos agora a complexidade das mensagens do algoritmo. Distinguimos duas fases no algoritmo.

1. A primeira fase consiste nos primeiros n ciclos em que os processadores estão enviando mensagens "é candidato" e propagando-as pelo anel. Nesta fase, cada processador recebe e envia uma mensagem por ciclo. Assim, $O(n^2)$ mensagens são enviadas na fase inicial.
2. A segunda fase consiste nos próximos n ciclos em que as mensagens "o líder é" percorrem o anel. Nesta fase, um processador continua enviando mensagens "é candidato" até que uma mensagem "o líder é" o alcance. Neste ponto, ele repassa a mensagem "o líder é" e pára. Assim, nesta fase, o líder irá enviar uma mensagem, o sucessor do líder irá enviar duas, seu sucessor, três e assim por diante. Desta forma, o número total de mensagens enviadas na segunda fase é $\sum_{i=0}^{n} i$,o que corresponde a $O(n^2)$.

Concluímos o que segue.

Teorema 11.1: *suponha uma rede em anel orientada e distribuída com n nodos, N, com identificadores distintos para cada nodo, mas sem um líder identificado. O algoritmo* RingLeader *encontra o líder de N usando um total de* $O(n^2)$ *mensagens. Além disso, a complexidade total das mensagens é* $O(n^2)$.

Prova: vimos que $O(n^2)$ mensagens são enviadas. Uma vez que cada mensagem usa $O(1)$ palavras, a complexidade total das mensagens é $O(n^2)$. ■

É possível melhorar a complexidade da eleição do líder em um anel modificando o algoritmo RingLeader de maneira que o mesmo requeira menos mensagens. Exploramos duas possibilidades de modificações nos Exercícios C-11.2 e C-11.3.

Eleição do líder assíncrona

O Algoritmo 11.2 (RingLeader) foi descrito e analisado em um modelo síncrono. Além disso, estruturamos os ciclos de maneira a ter um ciclo de recebimento de mensagens, um de processamento de mensagem e um de envio. Na verdade, esta é a estrutura clássica de um algoritmo distribuído síncrono.

Em um algoritmo assíncrono, não podemos mais assumir que os processadores se movem em "passos sincronizados". Em vez disso, o processamento é determinado por eventos, não por pulsos de um relógio. O algoritmo descrito para a eleição do líder em um anel trabalha em um modelo síncrono. A correção do algoritmo não pode mais depender do processamento síncrono dos processadores; só pode depender de que cada processador receba as mensagens de seu predecessor na mesma seqüência com que são enviadas. Esta condição vale para um modelo assíncrono.

Resumiremos nosso estudo sobre a eleição de um líder em uma rede com o seguinte teorema.

Teorema 11.2: *dado um anel orientado com n processadores, o algoritmo* RingLeader *executa a eleição do líder com complexidade de mensagens* $O(n^2)$. *Para cada processador, o tempo local de computação é* $O(n)$, *e o gasto de memória é* $O(1)$. *O algoritmo* RingLeader *trabalha tanto no modelo síncrono como no assíncrono. No modelo síncrono, o tempo total de execução é* $O(n)$.

A eleição do líder parece um problema artificial em um primeiro momento, mas na verdade tem muitas aplicações. Sempre que queremos calcular uma função global sobre um subconjunto dos nodos de uma rede, basicamente teremos de executar a eleição de um líder. Por exemplo, o cálculo da soma dos valores armazenados nos computadores conectados no anel, ou o mínimo e o máximo de tais valores, pode ser solucionados por um algoritmo similar ao algoritmo da eleição do líder descrito anteriormente.

Claro que os processadores não estão sempre conectados em anel. Então, na próxima seção, exploraremos a situação em que os processadores estão conectados em uma árvore (livre).

11.2.2 Eleição do líder em uma árvore

Se a rede for uma árvore (livre), a eleição do líder é muito mais simples do que em um anel, uma vez que uma árvore tem um ponto de início natural para as computações: os nodos externos.

Eleição assíncrona de um líder em uma árvore

O algoritmo para eleição de um líder em uma árvore é apresentado no Algoritmo 11.5. Novamente, escolhemos o processador com menor identificador como líder. O algoritmo assume que cada processador é capaz de executar, em intervalos constantes, uma ***checagem de mensagens*** sobre um aresta incidente para verificar se uma mensagem chegou por aquela aresta.

A idéia principal do algoritmo TreeLeader é usar duas fases. Na ***fase de acumulação***, os identificadores fluem dos nodos externos da árvore. Cada nodo mantém o identificador mínimo ℓ recebido de seus vizinhos, e após receber identificadores de todos os seus vizinhos menos um, envia o identificador ℓ para este vizinho. Em um ponto, um nodo recebe identificadores de todos os seus vizinhos. Este nodo, chamado de ***nodo de acumulação***, determina o líder. Demonstramos a fase de acumulação na Figura 11.4. Uma vez que o líder está identificado, iniciamos a ***fase de broadcast***. Na fase de *broadcast*, o nodo de acumulação divulga o identificador do líder para os nodos externos. Observe que pode ocorrer um laço onde dois nodos adjacentes se tornam nodos de acumulação. Neste caso, eles divulgam o identificador para suas respectivas "metades" da árvore. Qualquer nodo da árvore, mesmo um nodo externo, pode ser um nodo acumulador, dependendo da velocidade relativa dos processadores.

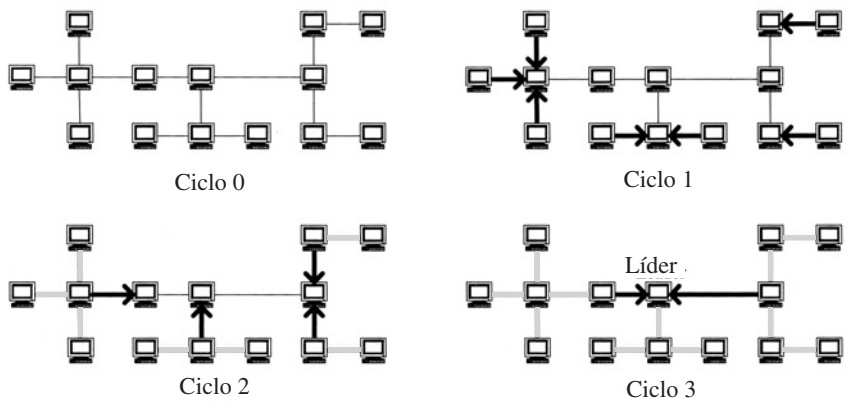


Figura 11.4 Ilustração do algoritmo para determinar o líder em uma árvore. Demonstramos apenas a fase de acumulação. Arestas que ainda não transportaram mensagens são mostradas com linhas finas. Arestas com mensagens fluindo naquele ciclo são desenhadas como setas grossas. Linhas cinza denotam arestas que transportaram mensagens em um ciclo anterior.

Algoritmo TreeLeader(*id*)
  Entrada: identificador único, *id*, do processador que está executando o algoritmo.
  Saída: o menor identificador dos processadores da árvore.
  {Fase de acumulação}
  seja d o número de vizinhos do processador *id* $\{d \geq 1\}$
  $m \leftarrow 0$ {contador de mensagens recebidas}
  $\ell \leftarrow id$ { líder provisório }
  repita
    {início de um novo *ciclo*}
    para cada vizinho *j* **faça**
      verifique se chegou mensagem do processador *j*

```
      se uma mensagem M = [Candidato é i] originária de j chegou, então
        ℓ ← min{i,ℓ}
        m ← m + 1
  até m ≥ d – 1
  se m = d então
    M ← [Líder é ℓ]
    para cada vizinho j ≠ k faça
      envie mensagem M para o processador j
    retorna M              {M é uma mensagem "o líder é"}
  senão
    M ← [Candidato é ℓ]
    envia M para o vizinho k que ainda não enviou mensagem
  {Fase Broadcast}
  repita
    {início de um novo ciclo}
    verifique se chegou mensagem do processador k
    se chegou mensagem M originária de k, então
      m ← m + 1
      se M = [Candidato é i] então
        ℓ ← mi§n{i,ℓ}
        M ← [Líder é ℓ]
        para cada vizinho j faça
          envie uma mensagem M para o processador j
      senão
        {M é uma mensagem "o líder é"}
        para cada vizinho j ≠ k, faça
          envie mensagem M para o processador j
  até m = d
  retorne M     {M é uma mensagem "o líder é"}
```

Algoritmo 11.5 Algoritmo para calcular o líder em uma árvore de processadores. Na versão síncrona do algoritmo, todos os processadores iniciam o ciclo ao mesmo tempo até parar.

Eleição síncrona do líder em uma árvore

Na versão síncrona do algoritmo, todos os processadores iniciam o ciclo ao mesmo tempo. Assim, as mensagens na fase de acumulação fluem para o nodo de acumulação da mesma forma que o inverso das ondas em uma lagoa depois de jogarmos uma pedra nela. Igualmente, na fase de *broadcast*, as mensagens se propagam a partir do ponto de acumulação na direção certa das ondas provocadas pela pedra na lagoa. Isto é, na fase de acumulação, as mensagens se dirigem para o centro da árvore e, na fase de *broadcast*, se dirigem para fora.

O ***diâmetro*** de um grafo é o comprimento do maior caminho entre dois nodos do grafo. Para uma árvore, o diâmetro é obtido pelo caminho entre dois nodos externos. É interessante observar que o número de ciclos na versão síncrona do algoritmo corresponde exatamente ao diâmetro da árvore.

Desempenho dos algoritmos de eleição do líder em uma árvore

Analisar a complexidade das mensagens do algoritmo de eleição do líder em uma árvore é relativamente simples. Durante a fase de acumulação, cada processador envia uma mensagem "o candidato é". Durante a fase de *broadcast*, cada processador envia ao menos uma mensagem "o lider é". Cada mensagem tem tamanho $O(1)$. Assim, a complexidade das mensagens é $O(n)$.

Estudaremos agora o tempo de execução local do algoritmo síncrono a partir da premissa de que ignoramos o tempo gasto esperando o início de um ciclo. O algoritmo para o processador i consome tempo $O(d_iD)$, onde d_i é o número de vizinhos do processador i, e D é o diâmetro da árvore. O processador i também usa espaço de memória d_i para controlar os vizinhos que já mandaram mensagens.

Resumimos nosso estudo sobre a eleição de um líder em uma árvore no seguinte teorema.

Teorema 11.3: *dada uma árvore (livre) com n nodos e com diâmetro D, o algoritmo* TreeLeader *executa a eleição do líder com complexidade de mensagens $O(n)$. Este algoritmo trabalha tanto no modelo síncrono como no assíncrono. No modelo síncrono, para cada processador i, o tempo local de computação é $O(d_iD)$ e o gasto local de memória é $O(d_i)$, onde d_i é o número de vizinhos do processador i.*

11.2.3 Pesquisa em largura

Suponhamos que dispomos de uma rede de processadores conectados e tenhamos identificado um vértice específico s desta rede como sendo um nodo-fonte. Na Seção 6.3.3, discutimos um algoritmo centralizado para pesquisa em largura sobre um grafo G iniciando em um nodo-fonte s. Nesta seção, descrevemos algoritmos distribuídos para resolver esse problema.

Pesquisa em largura síncrona

Descreveremos inicialmente um algoritmo simples para pesquisa em largura. A idéia central deste algoritmo é atuar em "ondas" que se propagam para fora a partir de um vértice-fonte s para construir uma árvore de pesquisa em largura, camada por camada, de cima para baixo. A sincronização dos processadores irá nos beneficiar enormemente, neste caso, permitindo que o processo de propagação seja inteiramente coordenado.

Iniciamos o algoritmo identificando s como um "nodo externo" da árvore de pesquisa em largura atual, que é composta apenas por s neste momento. Então, a cada ciclo, cada nodo externo v envia uma mensagem para todos seus vizinhos que ainda não contataram v, informando que v gostaria de transformar os mesmos em seus filhos na árvore de pesquisa em largura. Esses nodos respondem assumindo v como seu pai, se já não tiverem escolhido outro nodo como pai.

Ilustramos este algoritmo na Figura 11.6 e fornecemos o pseudocódigo correspondente no Algoritmo 11.7.

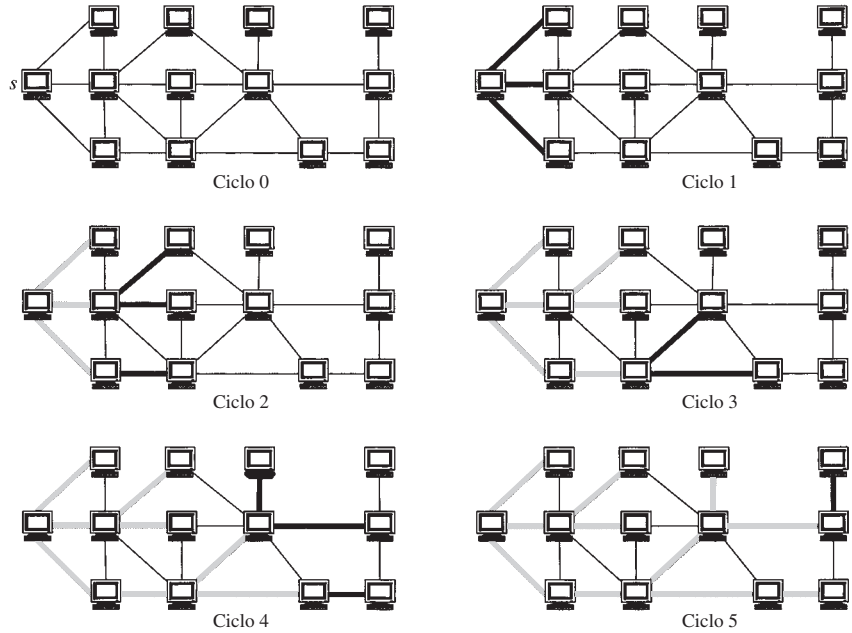


Figura 11.6 Ilustração do algoritmo sincronizado de pesquisa em largura. As arestas grossas são as da árvore de pesquisa em largura, e as arestas pretas e grossas são as que foram selecionadas no ciclo corrente.

Algoritmo SyncronousBFS(v, s):
  Entrada: o identificador v do nodo (processador) que está executando o algoritmo e o identificador s do nodo inicial da árvore de pesquisa em largura.
  Saída: para cada nodo v, seu pai na árvore de pesquisa em largura enraizada em s.
  repita
    {início de um novo ciclo}
    se $v = s$ **ou** v recebeu uma mensagem de um de seus vizinhos, **então**
      faça o pai(v) ser o nodo que requisitou v para ser seu filho (ou **null** se $v = s$)
      para cada nodo w adjacente a v que não contatou v ainda, **faça**
        envie uma mensagem para w solicitando que w se torne filho de v
  até $v = s$ ou v tenha recebido uma mensagem

Algoritmo 11.7 Algoritmo síncrono para computar uma árvore de pesquisa em largura em uma rede conectada de processadores.

Analisar o algortimo SinchronousBFS é simples. Em cada ciclo, propagamos um nível da árvore de pesquisa em largura. Assim, o tempo de execução desse algoritmo é proporcional à profundidade dessa árvore. Além disso, enviamos, quanto muito, uma mensagem para cada aresta em cada direção durante toda a computação. Assim, o número de mensagens enviadas em uma rede de n nodos e m arestas é $O(n + m)$. Desta forma, esse algoritmo é bastante eficiente. Entretanto, o mesmo faz uso considerável do sincronismo entre os processadores.

Pesquisa em largura assíncrona

Podemos tornar o algoritmo de pesquisa em largura assíncrono ao custo de mensagens adicionais para manter a coordenação. Além disso, é necessário que cada processador conheça o número total de processadores na rede.

Em vez de delegar a relógios sincronizados a tarefa de determinar cada ciclo da computação, teremos agora o nodo s enviando uma mensagem de "pulso" que irá disparar o próximo ciclo nos demais processadores. Usando essa técnica de pulsos, ainda podemos ter a propagação da computação nível a nível. O processo de pulso é como uma respiração – existe uma ***fase de pulso descendente***, em que o sinal é passado árvore abaixo a partir da raiz s, e uma ***fase de pulso ascendente***, em que o sinal é combinado a partir dos nodos externos voltando pela árvore até a raiz s.

Os processadores dos nodos externos podem ir de um ciclo para outro apenas se tiverem recebido um novo pulso descendente de seus respectivos pais. Da mesma forma, a raiz s não irá disparar um novo pulso descendente enquanto não houver recebido todos o pulsos ascendentes de seus filhos. Assim, podemos estar certos de que os processadores estão operando de uma forma grosseiramente sincronizada (na granularidade de ciclos). Descrevemos o algoritmo de pesquisa em largura assíncrono em detalhes no Algoritmo 11.8.

Algoritmo AsynchronousBFS(v, s, n):
  Entrada: o identificador v do processador que está rodando este algoritmo, o identificador s do nodo inicial do caminhamento da pesquisa em largura e o número n de nodos na rede.
  ***Saída*:** para cada nodo v, seu pai na árvore de pesquisa em largura enraizada em s.
  $C \leftarrow \emptyset$ {verifica o filho de v na árvore de pesquisa em largura}.
  Atribua para A o conjunto de vizinhos de v {candidatos a filhos de v na árvore de pesquisa em largura}
  repita
    {início de um novo ciclo}
    se parent(v) está definido ou $v = s$, **então**
      se parent(v) está definido, **então**
        aguarde por uma mensagem pulse-down do parent(v)
      se C não está vazio, **então**
        {v é um nodo interno da árvore de pesquisa em largura}
        envie uma mensagem pulse-down para todos os nodos de C
        aguarde por uma mensagem pulse-up de todos os nodos de C
      senão
        {v é um nodo externo na árvore de pesquisa em largura}
        para cada nodo u de A **faça**
          envie uma mensagem make-child para u
        para cada nodo u de A **faça**
          pegue a mensagem M de u e remova u de A
          se M é uma mensagem accept-child, **então**
            adicione u a C
      envie uma mensagem pulse-up para parent(v),
    senão
      {$v \neq s$ não tem pai ainda}
      para cada nodo w de A, **faça**

se w enviou para v uma mensagem make-child, **então**
remova w de A {w não é mais candidato a filho de v}
se parent(v) é indefinido, **então**
parent(v) $\leftarrow w$
envia uma mensagem accept-child para w
senão
envia uma mensagem reject-child para w
até (v tenha recebido uma mensagem done) **ou** ($v = s$ e tenha disparado pulse-down $n - 1$ vezes)
envie uma mensagem done para todos os nodos de C

Algoritmo 11.8 Algoritmo assíncrono para computar a árvore de pesquisa em largura em uma rede conectada de processadores. Este algoritmo deve ser executado simultaneamente em cada nodo v da rede. As mensagens possíveis são pulse-down, pulse-up, make-child, accept-child, reject-child e done.

Desempenho

O algoritmo de pesquisa em largura assíncrono é realmente mais complicado do que sua versão síncrona. A computação ainda ocorre por uma série de ciclos. Em cada ciclo, o nodo da raiz s envia uma mensagem de pulso descendente para a árvore de pesquisa em largura parcialmente construída. Quando essas mensagens encontram o nível externo da árvore, os nodos externos correntes tentam estendê-la de mais um nível, enviando mensagens "make-child" para os candidatos a filhos. Quando esses candidatos respondem, tanto aceitando como rejeitando o pedido, os processadores enviam uma mensagem de pulso ascendente que se propaga de volta até s. A raiz s então inicia todo o processo novamente.

Uma vez que a árvore de pesquisa em largura pode ter no máximo $n - 1$ nodos de altura máxima, o nodo s repete o envio dos pulsos $n - 1$ vezes apenas para ter certeza de que cada nodo na rede foi incluído na árvore. Assim, a sincronização ciclo a ciclo é completamente reslizada por troca de mensagens. Um nodo externo corrente não irá operar até receber um pulso descendente e não irá enviar um pulso ascendente até ter recebido uma resposta de todos os seus candidatos a filhos (que podem tanto rejeitar como aceitar o convite para se tornarem filhos). A árvore de pesquisa em largura cresce nível por nível como no algoritmo síncrono.

O número de mensagens necessárias para executar toda essa coordenação, entretanto, é maior do que o necessário para o algoritmo síncrono. Cada aresta na rede ainda lida com no máximo uma mensagem "make-child" e a respectiva resposta em cada direção. Assim, a complexidade de mensagens para aceitar e rejeitar solicitações "make-child" é $O(m)$, onde m é o número de arestas na rede. A cada ciclo do algoritmo, necessitamos ainda propagar todos as mensagens correspondentes aos pulsos descendentes e ascendentes. Uma vez que eles percorrem apenas as arestas da árvore, existem mais $O(n)$ mensagens de pulsos descendentes e descendentes enviados a cada ciclo. Dado que a fonte opera para $n - 1$ ciclos, existirão no máximo $O(n^2)$ mensagens criadas para lidar com a coordenação dos pulsos. Desta forma, a complexidade total das mensagens do algoritmo é $O(n^2 + m)$, onde m é o número de arestas na rede. Esse limite pode ser simplificado para $O(n^2)$, já que m é $O(n^2)$.

Assim, resumimos essa discussão com o teorema que segue.

Teorema 11.4: *dada uma rede G com n vértices e m arestas, podemos computar uma árvore de cobertura em largura em G em um modelo síncrono usando apenas $O(n^2)$ mensagens.*

No Exercício C-11.4, exploramos como melhorar o tempo de execução do algoritmo de pesquisa em largura para $O(nh + m)$, onde h é a altura da árvore.

11.2.4 Árvores de cobertura mínima

Lembramos que uma ***árvore de cobertura mínima*** (MST – *minimum spanning tree*) de um grafo balanceado é um subgrafo mínimo que é uma árvore e tem peso mínimo (Seção 7.3). Nesta seção, descrevemos um algoritmo distribuído eficiente para este problema. Antes de apresentarmos os detalhes, revisaremos primeiramente um algoritmo seqüencial eficiente para encontrar uma MST que é atribuído a Barůvka (Seção 7.3.3). O algoritmo inicia com todos os nodos de G em seu próprio componente conectado com (zero) arestas da árvore. Ele executa, então, uma série de ciclos para adicionar arestas da árvore para unir componentes conectados e, em função disso, construir uma MST. Como todos algoritmos de MST, ele é baseado no fato de que, para qualquer partição $\{V_1,V_2\}$ de vértices em dois subconjuntos não-vazios, se e é uma aresta de peso mínimo que une V_1 e V_2, então e pertence à árvore de cobertura mínima (veja o Teorema 7.10).

Por uma questão de simplicidade, vamos assumir que todos os pesos das arestas são únicos. Lembremos que, inicialmente, cada nodo do grafo está em seu próprio componente conectado (trivial) e que o conjunto inicial T, de arestas da árvore é vazio. No nível mais alto, o algoritmo de Barůvka executa as seguintes computações:

enquanto existe mais de um componente conectado definido por T, **faça**
  para cada componente conectado em paralelo C, **faça**
    escolha a menor aresta e que junta c com outro componente
    acrescente e ao conjunto T de arestas da árvore

O algoritmo de Barůvka executa as computações descritas para cada componente conectado em paralelo e, desta forma, encontra rapidamente as arestas da árvore. Observemos que na verdade, em cada ciclo, garantimos a união de cada componente conectado com pelo menos mais um. Assim, o número de componentes conectados decresce em um fator de 2 a cada ciclo (definido pela interação do laço **enquanto**), o que implica que o número total de ciclos é logarítmico em relação ao número de vértices.

O algoritmo de Barůvka em um ambiente distribuído

Vamos implementar uma versão do algoritmo de Barůvka no modelo síncrono (a versão assíncrona deste algoritmo é deixada para o Exercício C-11.7). Existem duas computações críticas necessárias em cada ciclo – temos de determinar todos os componentes conectados e necessitamos determinar a menor aresta de saída para cada um (lembramos que assumimos que os pesos das arestas são únicos; logo, a menor aresta de saída também é única) (veja a Figura 11.9).

Assumimos que, para cada nodo v, v armazena a lista das arestas de T que incidem em v. Assim, cada nodo v pertence a uma árvore, mas apenas v armazena informação sobre seus vizinhos em T. A cada ciclo, o algoritmo de eleição do líder em uma árvore da Seção 11.2.2 é usado duas vezes como método auxiliar:

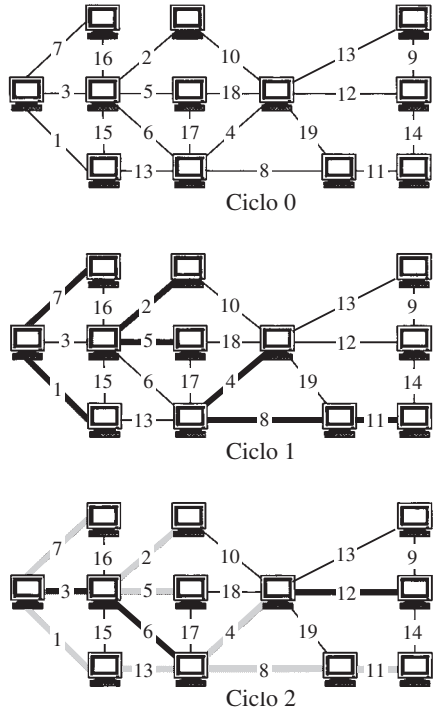


Figura 11.9 Ilustração da versão distribuída síncrona do algoritmo de Barůvka. As arestas grossas identificam arestas da MST, com as arestas pretas grossas sendo as escolhidas naquele ciclo.

1. para identificar cada componente conectado;
2. para encontrar, para cada componente conectado, a aresta de menor peso unindo um componente a outro.

Um ciclo funciona como segue. Para identificar os componentes conectados, computa-se a eleição do líder usando o identificador de cada nodo. Assim, cada componente fica identificado pelo menor identificador de seus nodos. Em seguida, cada nodo v determina a aresta de menor peso incidente em v, de maneira que os pontos finais de e estejam em componentes conectados diferentes. Se não existir tal aresta, usamos uma aresta fictícia associada a um peso infinito. A eleição do líder é então executada novamente usando as aresta associadas com cada vértice e seus pesos, o que resulta, para cada componente conectado C, na aresta de peso mínimo que conecta C a outro componente. Observemos que podemos detectar o fim do algoritmo no ciclo no qual computamos o peso da aresta de peso mínimo como sendo infinito.

Façamos n e m denotar o número de nodos e arestas, respectivamente. Para analisar a complexidade das mensagens da versão distribuída do algoritmo de Barůvka, observemos que $O(m)$ mensagens de tamanho fixo são enviadas a cada ciclo. Assim, uma vez que existem $O(\log n)$ ciclos, a complexidade total das mensagens é $O(m \log n)$. No Exercício C-11.6, examinamos uma maneira de melhorar a complexidade das mensagens para o algoritmo MST síncrono para $O(m + n \log n)$, e no Exercício C-11.7, de-

safiamos o leitor a encontrar um algoritmo assíncrono que mantenha a complexidade de mensagens em $O(m \log n)$.

11.3 Roteamento *broadcast* e *unicast*

Tendo estudado alguns dos algoritmos distribuídos fundamentais, vamos agora nos concentrar em dois problemas comuns em comunicação por rede – ***roteamento broadcast*** e ***roteamento unicast***. O roteamento *broadcast* é uma mensagem que é enviada de um processador para todos os outros processadores da rede. Essa forma de comunicação é usada quando a informação de um processador tem de ser compartilhada com todos os outros processadores da rede. O roteamento *unicast*, por outro lado, envolve a inicialização de estruturas de dados em uma rede de maneira a suportar comunicação ponto a ponto, em que um único processador tem uma mensagem que deseja que seja entregue para outro processador.

Quando discutirmos algoritmos de roteamento nas seções seguintes, iremos nos referir a um processador como sendo um ***roteador***. Lembramos que assumimos que a rede é fixa e não varia ao longo do tempo. Assim, nos restringimos ao estudo de algoritmos de roteamento estático. Também assumimos que as mensagens trocadas pelos roteadores têm tamanho constante, o que implica que a complexidade das mensagens é proporcional à quantidade de mensagens trocadas. Nesta seção, denotaremos com n e m o número de nodos e arestas de uma rede, respectivamente.

Iniciaremos nossa discussão pelo roteamento *broadcast*, e então cobriremos roteamento *unicast* e *multicast*. Os algoritmos apresentados nesta seção são versões simplificadas daqueles usados para roteamento de pacotes na Internet.

11.3.1 O algoritmo da inundação para roteamento *broadcast*

O ***algoritmo da inundação*** para roteamento *broadcast* é bastante simples e não requer praticamente nenhuma inicialização. Seu custo de roteamento, entretanto, é alto.

Um roteador s que deseja enviar uma mensagem M para todos os demais roteadores da rede começa por mandar M para seus vizinhos. Quando um roteador $v \neq s$ recebe uma mensagem M de um roteador adjacente u, v simplesmente reenvia M para todos os seus vizinhos, exceto para o próprio u. É claro que, sem modificações, esse algoritmo provoca um "laço infinito" de mensagens em qualquer rede que contenha um ciclo; uma rede sem ciclos, entretanto, permite algoritmos simples de roteamento, como exploraremos no Exercício C-11.5.

Inundação com um contador de saltos

Para evitar o problema dos laços infinitos, devemos associar alguma memória ou ***estado*** com os atores principais do algoritmo da inundação. Uma possibilidade é adicionarmos um ***contador de saltos*** para cada mensagem M e decrementá-lo cada vez que M for processado pelo roteador. Se o contador de saltos chegar a zero, então descartamos a mensagem à qual pertence. Se inicializarmos o contador de uma mensagem M com o diâmetro da rede, então podemos alcançar todos os roteadores e evitamos a criação de um número infinito de mensagens.

Inundação com seqüências de números

Outra possibilidade é armazenar uma tabela *hash* (ou outra estrutura de dados de dicionário eficiente) em cada roteador para manter o registro das mensagens que o roteador já processou. Quando um roteador x gerar uma mensagem *broadcast*, ele atribui a ela um ***número de seqüência*** k único. Assim, uma mensagem inundada é rotulada pelo par (x,k). Quando um roteador recebe uma mensagem rotulada com (x,k), ele verifica se (x,k) está em sua tabela. Em caso positivo, simplesmente descarta a mensagem. Caso contrário, adiciona (x,k) em sua tabela e reenvia a mensagem para seus vizinhos.

Esta abordagem certamente irá eliminar o problema do laço infinito, mas não é eficiente em termos de memória. Uma heurística comum é fazer cada roteador manter apenas o último número recebido de cada um dos roteadores da rede, partindo do princípio de que, se um roteador recebeu uma mensagem originada por x com um número de seqüência k, provavelmente já processou todas as mensagens enviadas por x com número de seqüência menor do que k.

Análise do algoritmo da inundação

Para analisar o algoritmos de inundação, devemos começar observando que praticamente não existe custo de inicialização, a não ser que cada roteador conheça o número n de roteadores na rede e talvez o diâmetro D da rede. Se usarmos a heurística de contadores de saltos, então não existe a necessidade de memória adicional nos roteadores. Se usarmos a heurística da seqüência de números, por outro lado, então a quantidade de memória necessária será $O(n)$, no pior caso. Em ambos os casos, o tempo esperado para que um roteador processe uma mensagem é proporcional ao número de vizinhos do roteador, uma vez que a pesquisa e a inserção em uma tabela *hash* têm tempo de execução $O(1)$ (veja a Seção 2.5.3).

Analisemos agora a complexidade das mensagens do algoritmo da inundação. Quando usamos a herística do contador de saltos, a complexidade das mensagens é $O((d_{\max} - 1)^D)$ no pior caso, onde $d_{\max}$ é o grau máximo dos roteadores em nossa rede. Quando usamos a heurística da seqüência de números, a complexidade das mensagens é $O(m)$, uma vez que enviamos as mensagens ao longo de todas as arestas da rede. Uma vez que m normalmente é muito menor do que $(d_{\max} - 1)^D$, números seqüenciais são normalmente preferidos em vez de contadores de saltos. Nos dois casos, entretanto, o algoritmo da inundação é eficiente apenas para as mensagens que pretendemos enviar para todos os outros roteadores da rede.

Entretanto, o algortimo da inundação seguramente envia uma mensagem M da sua fonte até o destino no menor número de saltos considerando modelo de rede sincronizado. Isto é, sempre encontra o menor caminho. Dado o *overhead* envolvido no algoritmo da inundação, entretanto, é interessante dispor de um algoritmo que possa rotear mensagens pelo caminho mais curto, de forma mais eficiente. O algoritimo que discutiremos em seguida faz isso executando algumas computações de inicialização para determinar as melhores rotas pela rede.

11.3.2 Algoritmo do vetor de distâncias para roteamento *unicast*

O primeiro algoritmo para roteamento *unicast* que discutiremos é uma versão distribuída de um algoritmo clássico para encontrar o menor caminho em um grafo atribuído a Bellman e Ford (Seção 7.1.2). Assumimos que, associado a cada aresta da rede e, existe um peso positivo $w(e)$ que representa o custo para enviar uma mensagem pela mesma. Por exemplo, o peso pode representar a "latência" média do canal de comunicação.

O problema de roteamento *unicast* é configurar as estruturas de dados nos nodos da rede para suportar o roteamento eficiente de uma mensagem do nodo de origem até o destino.

O algoritmo do ***vetor de distâncias*** sempre define rotas pelos menores caminhos. A idéia principal deste algoritmo é, para cada roteador x, armazenar um arranjo chamado de ***vetor de distâncias*** do roteador x que armazena o comprimento do melhor caminho conhecido entre x e qualquer outro roteador y da rede, denotado por $D_x[y]$, e a primeira aresta (conexão) de tal caminho, denotada por $C_x[y]$. Inicialmente, para cada aresta, atribuímos

$$D_x[y] = D_y[x] = w(x, y)$$

e

$$C_x[y] = C_y[x] = (x, y).$$

As demais entradas D_x são inicializadas com $+\infty$. Executamos, então, uma série de ciclos que iterativamente refinam os vetores de distâncias para encontrar possíveis melhores caminhos até que todos os vetores de distâncias armazenem a distância real para os demais roteadores.

Relaxamento distribuído

A inicialização do algoritmo do vetor de distâncias consiste em uma série de ciclos. Por vez, cada ciclo consiste em uma coleção de etapas de ***relaxamento***. No início de um ciclo, cada roteador envia seu vetor de distâncias para todos os seus vizinhos imediatos na rede. Após um roteador x ter recebido o vetor de distâncias corrente de todos os seus vizinhos, ele executa as seguintes computações locais para $n - 1$ ciclos:

para cada roteador w conectado a x **faça**
  para cada roteador y conectado na rede **faça**
    {*relaxamento*}
    se $D_w[y] + w(x,w) < D_x[y]$ **então**
      {uma rota melhor entre x e y por w foi encontrada}
      $D_x[y] \leftarrow D_w[y] + w(x,w)$.
      $C_x[y] \leftarrow (x,w)$.

Desempenho

Uma vez que cada vértice x envia um vetor de tamanho n para todos os seus vizinhos a cada ciclo, o tempo e a complexidade das mensagens de x para completar um ciclo é $O(d_x n)$, onde d_x é o grau de x, e n é o número de roteadores na rede. Lembramos que iteramos esse algoritmo para $n - 1$ ciclos. Na verdade, podemos melhorar isso para D

ciclos, onde D é o diâmetro da rede, uma vez que os vetores de distâncias não irão mudar após essa quantidade de ciclos.

Após a inicialização, o vetor de distâncias de um roteador x armazena as distâncias de x para os demais roteadores da rede, bem como a primeira aresta destes caminhos. Conseqüentemente, após a execução da inicialização, o algoritmo de roteamento é muito simples: se um roteador x receber uma mensagem destinada ao roteador y, envia y pela aresta $C_x[y]$.

Correção do algoritmo do vetor de distâncias

A correção do algoritmo do vetor de distâncias segue um argumento indutivo:

Lema 11.5: *ao final do ciclo i, cada vetor de distâncias armazena o menor caminho para cada outro roteador restrito a visitar no máximo i outros roteadores no caminho.*

Este fato se verifica no início do algoritmo, e os relaxamentos feitos em cada ciclo garantem que será verdade também após cada um dos ciclos.

Análise do algoritmo do vetor de distâncias

Vamos analisar a complexidade do algoritmo do vetor distância. Ao contrário do algoritmo da inundação, discutido na Seção 11.3.1, o algoritmo do vetor de distâncias tem um custo de inicialização significativo. O número total de mensagens trocadas em cada ciclo é proporcional a n vezes a soma de todos os graus da rede. Isso ocorre porque o número de mensagens enviadas e recebidas em um roteador x é $O(d_x n)$, onde d_x denota o grau de x. Assim, existem $O(nm)$ mensagens por ciclo. Desta forma, a complexidade total de mensagens para a inicialização do algoritmo do vetor de distâncias é $O(Dnm)$, o que corresponde a $O(n^2m)$ no pior caso.

Após a inicialização estar completa, cada roteador armazena um arranjo com $n - 1$ elementos, o que consome um espaço $O(n)$, e processa uma mensagem em tempo $O(1)$. Observamos que a necessidade de memória local combina com a do algoritmo da inundação quando usa a heurística da seqüência de números. Uma vez que o algoritmo do vetor de distâncias encontra os menores caminhos, da mesma forma que o algoritmo da inundação, mas fazendo isso durante a fase de inicialização, ele pode ser visto como aumentando os custos da inicialização para uma entrega de mensagens mais eficiente.

11.3.3 Algoritmo *link-state* para roteamento *unicast*

O último algoritmo estático que iremos abordar é uma versão distribuída do clássico algoritmo para determinar o menor caminho, atribuído a Dijkstra. Como na descrição do algoritmo do vetor de distâncias, assumimos que as arestas têm pesos positivos. Enquanto o algoritmo do vetor de distâncias executa sua inicialização em séries de ciclos que requerem comunicação apenas entre roteadores adjacentes, o algoritmo *link-state* computa um único ciclo de comunicação que requer várias comunicações globais por toda a rede.

O algoritmo ***link-state*** funciona da seguinte forma. A inicialização começa com cada roteador x enviando os pesos de suas arestas incidentes (isto é, o *status* de x) para todos os roteadores da rede, usando o algoritmo de roteamento da inundação com seqüência de números (que não requer nenhuma inicialização prévia; veja a Seção

11.3.1). Após essa fase de difusão, cada roteador conhece toda a rede e pode executar o algoritmo de menor caminho de Dijkstra para determinar o menor caminho partindo dele próprio para qualquer outro roteador y, bem como a primeira aresta $C_x[y]$ de tal caminho. Esse cálculo interno consome tempo $O(m \log n)$, usando a implementação *standard*, do algoritmo de *Dijkstra*, ou $O(n \log n + m)$, usando estruturas de dados mais sofisticadas (lembre-se da Seção 7.1.1).

Como no algoritmo do vetor de distâncias, a estrutura de dados construída pela inicialização de cada roteador ocupa espaço $O(n)$ e suporta o processamento de mensagens em tempo $O(1)$.

Vamos analisar agora a complexidade das mensagens da inicialização do algortimo *link-state*. Um total de m mensagens de tamanho constante são distribuídas, cada qual causando, por sua vez, o envio de m mensagens pelo algoritmo da inundação. Assim, a complexidade geral de mensagens da etapa de inicialização é $O(m^2)$.

Comparação dos algoritmos de roteamento *Broadcast* e *Unicast*

Comparamos o desempenho dos três algoritmos estáticos de roteamento discutidos nesta seção na Tabela 11.10. Observamos que não estão incluídos os tempos internos de computação que são requeridos de um roteador participando da fase de pré-processamento do cálculo da inicialização nesta tabela.

Algoritmo	Mensagens	Espaço local	Tempo local	Tempo de roteamento
Inundação c/ contador de saltos	$O(1)$	$O(1)$	$O(d)$	$O((d-1)^D)$
Inundação com seqüência de nºs	$O(1)$	$O(n)$	$O(d)$	$O(m)$
Vetor de distâncias	$O(Dnm)$	$O(n)$	$O(1)$	$O(p)$
Link-state	$O(m^2)$	$O(n)$	$O(1)$	$O(p)$

Tabela 11.10 Limites de desempenho assintótico para algoritmos estáticos de roteamento. Usamos a notação a seguir para a rede e os parâmetros de roteamento: n, número de nodos; m, número de arestas; d, grau máximo do nodo; D, diâmetro; e p, número de arestas no menor caminho. Observamos que p, d e D são todos menores do que n.

11.4 Roteamento *multicast*

Até então, estudamos dois tipos de algoritmos de roteamento para dois tipos de comunicação – roteamento *broadcast* e *unicast* – que podem ser vistos como dois extremos de um espectro. Em algum lugar no meio deste espectro, está o ***roteamento multicast***, que envolve a comunicação com um subconjunto dos hospedeiros de uma rede, chamados de ***grupo multicast***.

11.4.1 Caminhamento reverso

O algoritmo de caminhamento reverso (RPF – *reverse path forwarding*) adapta o algoritmo da inundação de roteamento *brodcast* para roteamento *multicast*. Ele é projetado para trabalhar em conjunto com as tabelas de roteamento de menor caminho existentes disponíveis em cada roteador, de maneira a divulgar uma mensagem *multicast* pela árvore de menor caminho a partir de uma fonte.

Este método inicia com um hospedeiro que deseja enviar uma mensagem para um grupo g. O hospedeiro envia uma mensagem para seu roteador local s de maneira que s envie a mensagem para todos os seus roteadores vizinhos. Quando um roteador x recebe de um de seus vizinhos y uma mensagem *multicast* que se originou no roteador s, x examina sua tabela de roteamento local para verificar se y está no caminho mais curto de x até s. Se y não está no caminho mais curto até s, então (assumindo que todas as tabelas de roteamento estão corretas e consistentes) a ligação de y até x não está na árvore de menor caminho de s. Assim, neste caso, x simplesmente descarta o pacote enviado de y e envia de volta uma mensagem ***de corte***, incluindo o nome da fonte s e o grupo g. Essa mensagem de corte diz para y parar de enviar mensagens *multicast* de s destinadas para g (e, se possível, não enviar nenhuma mensagem *multicast* de s por esta ligação, não importando que grupo seja). Se, por outro lado, y está no caminho mais curto de x para s, então x replica a mensagem e a envia para todos os seus roteadores vizinhos, excetuando o próprio y. Neste caso, a ligação de y para x está na árvore de menor caminho de s (novamente, assumindo que todas as tabelas de roteamento são corretas e consistentes).

Este modo de comunicação *broadcast* se estende de s pela árvore de menor caminho T de s até fluir por toda a rede. O fato de o algoritmo enviar uma mensagem *multicast* para todos roteadores na rede é um desperdício se apenas uma pequena fração dos roteadores da rede tiverem clientes que desejam receber mensagens *multicast* que são enviadas para o grupo g.

Para lidar com esse desperdício, o algoritmo RPF provê um tipo adicional de mensagem de corte. Especificamente, se um roteador x, em um nodo externo de T, determinar que em sua rede local não há clientes interessados em receber mensagens do grupo g, então x envia uma mensagem de corte para seu pai y em T, indicando para y parar de enviar suas mensagens de s para o grupo g. Esta mensagem, na verdade, diz para y remover o nodo externo x da árvore T.

Cortes

Essas mensagem de corte podem, é claro, ser enviadas por vários dos nodos externos de T em paralelo; sendo assim, podemos ter vários nodos externos de T removidos ao mesmo tempo. Além disso, a remoção de alguns dos nodos externos de T pode levar à criação de novos nodos externos. Desta forma, o algortimo RPF testa cada roteador x continuamente para verificar se ele se torna um nodo externo de T e, se for o caso de x, testa se há algum cliente hospedado na sua rede local que esteja interessado em receber mensagens *multicast* oriundas do grupo g. Novamente, se não existirem tais clientes, então x envia para seu pai em T uma mensagem de corte relativa às mensagens *multicast* oriundas de s para o grupo g. Esses cortes de nodos externos continuam até que todos os nodos externos remanescentes em T tenham clientes que desejam receber mensagens *multicast* para o grupo g. Neste ponto, o algoritmo RPF alcançou um estado estável.

Este estado estável, entretanto, não pode ser mantido para sempre, pois pode haver alguns clientes hospedeiros em nossa rede que desejam começar a receber mensagens do grupo g. Isto é, podemos ter pelo menos um cliente desejando ***unir-se*** a grupo g. Uma vez que o algoritmo RPF não provê nenhuma forma explícita para um cliente se unir a um grupo, a única forma de h começar a receber mensagens *multicast* para o grupo g é se o roteador de h, x, estiver recebendo esses pacotes. Mas se x foi cortado de T, então isso não vai ocorrer. Por esta razão, um componente adicional do algoritmo RPF faz com que os cortes armazenados em um nodo da Internet tenham sua validade expirada após um certo período de tempo.

Quando esse período se esgota, digamos para um corte proveniente de um roteador z para o roteador x que mantém uma mensagem de corte prévia de z, então x passa a enviar pacotes *multicast* para o grupo g a z. Entretanto, se um roteador realmente não pretende receber ou processar certos tipos de mensagens *multicast*, então ele deve continuamente informar a seus vizinhos sobre sua vontade, usando mensagens de corte. Por esta razão, o algoritmo RPF não é um método eficiente de roteamento *multicast*.

Desempenho

Em termos de eficiência de mensagens, o RPF inicialmente envia $O(m)$ mensagens, onde m é o número de conexões (arestas) na rede. Após a primeira onda de cortes se propagar pela rede, entretanto, a complexidade do *multicast* se reduz a $O(n)$ mensagens por mensagem *multicast*, onde n é o número de roteadores. E isso pode se reduzir ainda mais à medida que as mensagens de corte são processadas. Em termos de armazenamento adicional nos roteadores, o algoritmo RPF não é tão eficiente, pois requer que cada roteador armazene todas as mensagens de corte que recebeu até que fiquem vencidas. No pior caso, então, um roteador x pode ter de armazenar $O(|\mathcal{S}| \cdot |\mathcal{G}| d_x)$ mensagens de corte, onde $\mathcal{S}$ é o conjunto de fontes, $\mathcal{G}$ é o conjunto de grupos e d_x é o grau de x na rede. Assim, como já observamos, o algoritmo RPF não é eficiente.

11.4.2 Árvores centradas

Um algoritmo *multicast* com eficiência de mensagens melhor do que a do algortimo do caminhamento reverso é o método das ***árvores centradas***. Neste algoritmo, escolhemos, para cada grupo g, um único roteador z da rede que irá atuar como nodo "central" ou "marcador de encontros" para o grupo g. O roteador z forma o nodo da raiz da árvore *multicast* T para g, o qual é usado para enviar mensagens para roteadores que são parte deste grupo. Qualquer fonte que deseja mandar uma mensagem *multicast* para o grupo g primeiramente envia uma mensagem para o centro z. Na forma mais simples do algoritmo, essa mensagem prossegue até z, e uma vez recebida por z, este se encarrega de enviar a mensagem para todos os nodos de T. Assim, cada roteador x representado por um nodo em T sabe que, se x recebe uma mensagem *multicast* para g de seu pai em T, então x replica essa mensagem e a envia para todos os seus vizinhos que correspondem a seus filhos em T. Igualmente, na forma mais simples do método das árvores centradas, se x recebe uma mensagem *multicast* de qualquer outro vizinho diferente de seu pai em T, então envia esta mensagem árvore acima para z. Tal mensagem é proveniente de uma fonte e deve ser enviada para z antes de ser distribuída para T.

Como mencionado anteriormente, devemos manter explícita a árvore *multicast* T de cada grupo g. Assim, podemos fornecer uma forma de os roteadores se ***unirem*** ao

grupo g. Uma operação de junção para um roteador x inicia por x enviando uma mensagem de junção para o nodo central z. Qualquer outro roteador y que receber uma mensagem de junção de um vizinho t procura verificar qual y vizinhos de u está no menor caminho para z, e então y cria e armazena um registro interno indicando que agora y tem um filho t e um pai u na árvore T. Se y já estiver na árvore T (ou seja, já existe um registro indicando que u é pai de y em T), então isso completa a operação de união – o roteador x é agora um nodo externo conectado na árvore T. Se y ainda não estiver na árvore T, entretanto, então y propaga a mensagem de união para seu (novo) pai u em T (veja a Figura 11.11).

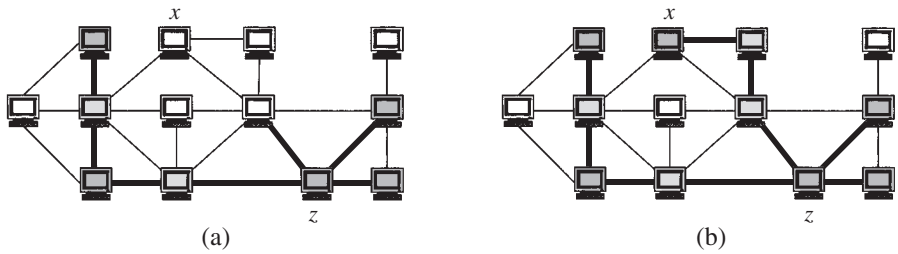


Figura 11.11 Ilustração do protocolo das árvores centradas. (a) Uma árvore *multicast* centrada em z, com as arestas grossas sendo as arestas da árvore, os nodos cinza-escuro sendo os membros do grupo e os nodos cinza-claro sendo os roteadores que simplesmente propagam as mensagens; (b) a árvore *multicast* após a junção de x. Observamos que x pode se unir usando apenas uma aresta e nenhum nodo intermediário adicional, mas o caminho para z não será o menor.

Desempenho

Para roteadores que desejam deixar o grupo g, tanto por mensagens ***de partida*** explícitas ou porque um registro de junção teve seu prazo de validade expirado, nós essencialmente invertemos a ação executada para a junção. Em termos de eficiência, o método das árvores centradas envia mensagens apenas ao longo de T, a árvore *multicast* para o grupo g; logo, sua complexidade de mensagens é $O(|T|)$ para cada *multicast*. Além disso, como cada roteador armazena apenas a estrutura local de T, o número máximo de registros que precisam ser armazenados na estrutura local de T é $O(|\mathcal{G}|d_x)$, onde $\mathcal{G}$ é o conjunto dos grupos e d_x é o grau de x na rede. Conseqüentemente, O algoritmo da árvore centrada tem a vantagem da eficiência, que vem do fato de sempre enviar mensagens *multicast* para um grupo g por uma única árvore T. Essa árvore, entretanto, pode não ser a árvore mais eficiente para enviar tal mensagem.

11.4.3 Árvores de Steiner

O principal objetivo do algoritmo das árvores de Steiner é otimizar o custo total de todas as arestas que estamos usando para construir a árvore *multicast* para um grupo g. Formalmente, fazemos $G = (V, E)$ denotar nossa rede, com V sendo o conjunto de roteadores e definindo uma métrica de custo, $c(e)$, para as arestas de G. Além disso, assumimos um dado subconjunto S de V que indica os roteadores de nossa rede que pertencem ao grupo g. A ***árvore de Steiner*** T para S é a árvore de custo mínimo total que conecta todos os nodos em S. Observamos que, se $|S| = 1$, então T é um nodo único, que se $|S| = 2$, então T é a distância mínima entre os dois nodos em S, e se $|S| = |V|$, então T é a árvore de cobertura mínima para T. Assim, para esses casos especiais, pode-

mos construir facilmente a árvore ótima T unindo os nodos de S, e podemos usar essa árvore para todos os *multicasts* do grupo g, como no algoritmo da árvore centrada. Infelizmente, resolver o caso geral do problema da árvore de Steiner é um problema ***NP***-completo (Seção 13.2.1); por esta razão, devemos tentar empregar algumas heurísticas de algoritmos que podem aproximar a árvore de Steiner.

O algoritmo de aproximação da árvore de Steiner que descrevemos é baseado no algoritmo da árvore de cobertura mínima descrito anteriormente (na Seção 11.2.4). Ele usa informações já disponíveis em cada roteador da rede e é conhecido como o algoritmo da ***rede de distâncias***, pois opera em um grafo G', derivado de G, conhecido como a rede de distâncias de S. O grafo G' é formado tomando-se os vértices de S como seus nodos e conectando cada par de tais nodos com uma aresta. O custo $c'(e)$ de uma aresta $e = (v,w)$ de G' é exatamente a distância entre v e w em G, ou seja, o comprimento do menor caminho de v até w em G (assumimos que G seja não-dirigido). O algoritmo específico é o seguinte:

1. Construa uma representação para a rede de distâncias G' de S.
2. Encontre uma árvore de cobertura mínima (MST) T_M em G'.
3. Traduza T_M em uma árvore T_A,tomando o menor caminho de v até w para cada aresta (v,w) em T_M.
4. Encontre o MST T_D no subgrafo de G induzido pelos nodos de T_A.
5. Remova de T_D qualquer caminho que leve a nodos externos que não estão em S.

Os últimos dois passos provêm alguns melhoramentos adicionais na árvore T_A, a qual já é uma boa aproximação da árvore T_S de Steiner.

Lema 11.6: $|T_A| \leq |T_S|(2 - 2/k)$, *onde* $k = |S|$.

Prova: consideremos a árvore de Steiner T_S, que conecta otimamente os nodos de S em G. Consideremos ainda a trajetória W, formada tomando-se um caminhamento contínuo (em profundidade) ao redor da borda de T_S. Isto é, imagine que T_S seja o mapa de um conjunto de estradas que percorremos nos mantendo sempre do lado direito da estrada. Assim, $|W| = 2|T_S|$. Subdivida W no conjunto de k caminhos conectando os pares de nodos de S que são visitados consecutivamente no caminho W, subdividindo W apenas na primeira visita que faz a cada nodo (veja a Figura 11.12a). Faça P denotar o conjunto de $k - 1$ caminhos que restam após a remoção do caminho de maior custo deste conjunto. Assim, $|P| \leq (2 - 2/k)\ |T_S|$. Faça P' denotar o conjunto de caminhos formados pela substituição de cada caminho p de P, que vai de um nodo v para um nodo w de S, com o menor caminho de v até w. Assim $|P'| \leq |P|$. Finalmente, faça T' denotar o conjunto de arestas na rede de distâncias formada pela substituição de cada caminho em P' por sua aresta correspondente na rede de distâncias de S (veja a Figura 11.12b). Observamos que T' é uma árvore de cobertura na rede de distância para S. Logo, $|T_A| \leq |T'|$. Conseqüentemente, $|T_A| \leq |T_S|(2 - 2/k)$. ■

Uma vez que T_D não é pior do que T_A, o lema acima implica que $|T_D| \leq (2 - 2/k)|T_S|$.

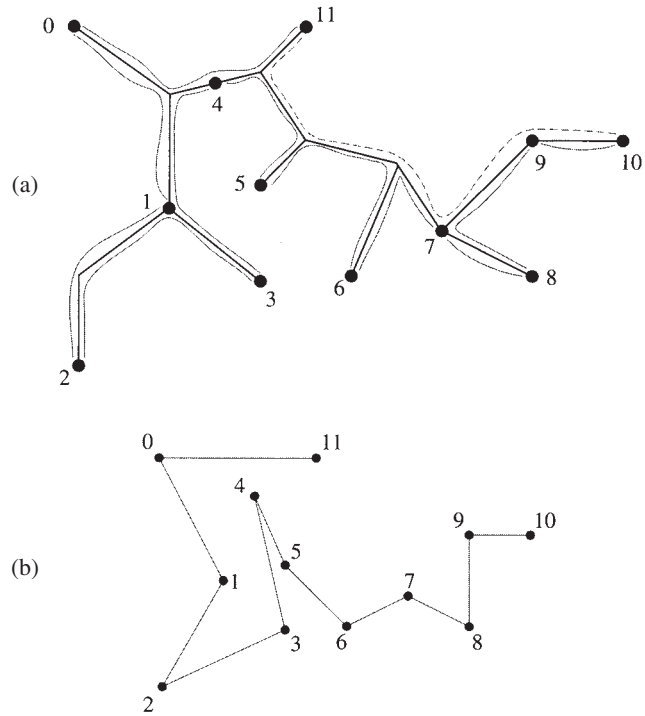


Figura 11.12 Ilustração da prova do Lema 11.6: (a) a árvore de Steiner, T_S, com os vértices em S destacados e a trajetória W particionada em $k = 12$ caminhos. O caminho de maior custo é mostrado tracejado. (b) A árvore T' na rede de distância.

Análise

Além de produzir uma boa aproximação para a árvore de Steiner, o algoritmo da rede de distância também é fácil de implementar em uma configuração de rede. A própria rede de distância é implicitamente representada por um vetor de distâncias que pode ser armazenado em cada roteador da rede (após o algoritmo do estado da ligação, como o do vetor de distâncias). Assim, se cada roteador conhece quais os roteadores que pertencem ao grupo g, então não existem computações adicionais necessárias para compor a rede de distância G'. Encontrar uma árvore de cobertura mínima em G', então, pode ser feito usando o algoritmo de cobertura mínima distribuído descrito na Seção 11.2.4, onde além dos nodos examinarem apenas as conexões adjacentes em um único ciclo, temos cada nodo de S considerando todos menores caminhos para os outros nodos de S. Mas se cada nodo armazena um vetor de distâncias para todos os outros nodos da rede, não necessitamos questionar os outros nodos para determinar a distância para os mesmos. Em cada ciclo, necessitamos apenas determinar, para cada nodo x de S, o menor caminho para outro nodo de S que não seja o mesmo conectado ao componente x. Podemos fazer isso se x enviar mensagens em G para os outros nodos de S, ordenados por sua distância, perguntando aos outros nodos se ele está no mesmo componente conectado como x. Isto é, cada etapa do algoritmo distribuído pode ser executada roteando mensagens em G. Como cada mensagem irá requerer no máximo D saltos, onde D é o diâmetro de G, isso implica que podemos encontrar T_A usando no máximo $O(k^2D)$ mensagens. Se adicionalmente quisermos computar a árvore T_D, en-

tão precisamos transformar um algoritmo adicional MST no próprio G, o que pode ser feito usando $O(k_2D)$ mensagens. Assim, esse algoritmo de aproximação do problema da árvore de Steiner tem uma complexidade de mensagens eficiente.

Apesar disso, esse algoritmo se aplica apenas para grupos que são estáveis no tempo, uma vez que qualquer modificação no grupo requeira que construamos uma nova árvore *multicast* para aquela árvore desde o início. Notamos, entretanto, que cada um dos algoritmos de *multicast* que estudamos tem seus problemas. Na verdade, ainda existe um trabalho considerável a ser feito para obtermos novos algoritmos de roteamento *multicast* mais eficientes.

11.5 Exercícios

Reforço

R-11.1 Desenhe uma figura que ilustre as mensagens enviadas a cada ciclo do algoritmo síncrono de eleição do líder em uma rede em anel com 10 nodos.

R-11.2 Desenhe uma figura que ilustre as mensagens enviadas em cada ciclo do algoritmo síncrono de eleição do líder em uma árvore de 10 nodos. Projete o processamento de maneira que dois nodos possam ser líderes.

R-11.3 Desenhe uma figura que ilustre como o algoritmo BFS síncrono trabalha. Sua figura deve resultar em uma árvore BFS com pelo menos cinco níveis.

R-11.4 Desenhe uma figura que ilustre como o algoritmo BFS assíncrono trabalha. Sua figura deve resultar em uma árvore BFS com pelo menos três níveis e deve mostrar tanto ações de pulso ascendente como descendente.

R-11.5 Desenhe uma rede de computadores que faça com que o algoritmo da árvore de cobertura mínima distribuído de Barůvka execute por quatro ciclos. Mostre as arestas escolhidas em cada ciclo.

R-11.6 Forneça a descrição em pseudocódigo do algoritmo que um computador execute para rotear uma mensagem de inundação que use um contador de saltos.

R-11.7 Forneça a descrição em pseudocódigo do algoritmo que um computador execute para rotear uma mensagem de inundação que use uma seqüência de números.

R-11.8 Forneça uma descrição em pseudocódigo de um algoritmo que um computador execute para rotear uma mensagem *unicast* após já ter executado a inicialização dos algoritmos *link-state* ou vetor de distâncias.

R-11.9 Desenhe uma figura que ilustre como o algoritmo do vetor de distâncias trabalha em uma rede de pelo menos 10 nodos. Sua ilustração deve mostrar uma computação que leve pelo menos três ciclos para estabilizar.

R-11.10 Exatamente quantas mensagens são enviadas para executar a inicialização do algoritmo *link-state* em uma rede com 10 nodos e 20 arestas, assumindo que cada nodo já conhece o estado de suas arestas incidentes.

R-11.11 Desenhe uma figura que ilustre operações de corte no algoritmo *multicast* do caminhamento reverso.

R-11.12 Desenhe uma figura que mostre que a abordagem das árvores centradas para roteamento *multicast* pode usar duas vezes mais arestas do que a abordagem da árvore de Steiner.

R-11.13 Desenhe uma figura que mostre que o algoritmo do caminhamento reverso para roteamento *multicast* pode usar mais do que a abordagem da árvore de Steiner.

Criatividade

C-11.1 Estenda o algoritmo da eleição do líder para o caso em que a rede é um ciclo não-orientado e as mensagens podem ser passadas em qualquer direção.

Dica: considere um ciclo não-orientado como a união de dois ciclos orientados.

C-11.2 Considere o problema da eleição de um processador líder em uma rede em anel de $n \geq 4$ processadores, cada um com um único identificador. Assumindo que todos os processadores conhecem o valor de n, projete um algoritmo síncrono aleatório cujo tempo esperado de execução seja $O(n)$ e cuja complexidade esperada de mensagens seja também $O(n)$.

Dica: explore o fato de que, se cada um dos $n \geq 4$ processadores decide de forma independente para enviar uma mensagem com probabilidade $4/n$, então o número esperado de processadores que envia mensagens é 4, e a probabilidade de nenhum processador enviar uma mensagem é menor do que 1/2.

C-11.3 Considere o problema da eleição de um processador líder em um anel de n processadores, cada um dos quais com um identificador único. Projete um algoritmo determinístico síncrono cujo tempo de execução e complexidade de mensagem seja $O(n \log n)$.

Dica: considere um algoritmo com $\log n$ fases, onde cada processador que, na fase i, pensa poder ser o líder envia uma mensagem "de sondagem" em cada direção que avança 2^i saltos e retorna se encontrar um processador com valor de *id* menor.

C-11.4 Projete um algoritmo de pesquisa em largura (BFS) para uma rede de n vértices e m arestas que tenha uma complexidade total de mensagens que seja $O(nh + m)$, onde h é a altura da árvore BFS.

Dica: estenda as mensagens de "pulso ascendente" e "pulso descendente" de maneira que a raiz s possa saber o ciclo exato no qual a árvore BFS está completamente formada.

C-11.5 Suponha que uma rede conectada G não contenha ciclos; isto é, G é uma árvore (livre). Descreva uma maneira de numerar os nodos de G de maneira que qualquer nodo x armazene apenas $O(1)$ informações, mas que possamos rotear uma mensagem do computador y para um computador z em G sem nenhum retorno.

C-11.6 Descreva um algoritmo distribuído assíncrono para encontrar uma árvore de cobertura mínima em uma rede balanceada de n vértices e m arestas tendo complexidade de mensagens $O(m + \log n)$.

Dica: considere que a vantagem vem do fato de primeiramente ordenarmos as arestas incidentes de cada vértice.

C-11.7 Descreva um algoritmo distribuído assíncrono para encontrar uma árvore de cobertura mínima em uma rede balanceada de n vértices e m arestas tendo complexidade de mensagens $O(m \log n)$.

Dica: considere o uso de uma estratégia "pulsante" que conta os ciclos que o algoritmo está executando.

C-11.8 Descreva uma modificação no algoritmo distribuído da árvore de cobertura mínima descrito em aula de maneira que não necessitemos assumir que os pesos das arestas são únicos.

Dica: descreva uma regra de desempate de maneira que sob esta regra a árvore de cobertura seja mínima.

C-11.9 Considere uma versão passiva dinâmica do algoritmo da inundação usando seqüência de números que seja capaz de se recuperar de partições na rede que ocorram enquanto o algoritmo está executando. A idéia principal desta abordagem é que cada roteador que inicia uma inundação armazene as mensagens velhas por uma quantidade de tempo maior do que qualquer falha de rede antecipável. Cada roteador armazena ainda a última seqüência de números de mensagens de inundação proveniente de qualquer outro roteador. Mas se um roteador x receber uma mensagem de inundação y com um número de seqüência i maior do que 1 e maior do que o número de seqüência j de x previamente armazenado recebido de y, então x inicia respondendo com uma mensagem de inundação que solicita que y reenvie sua mensagem $j + 1$ até $i - 1$. Mostre como a acomodação desse "conserto" pode resultar em um laço infinito de mensagens de inundação.

C-11.10 Considere uma versão do algoritmo da inundação que use uma seqüência de números ou contadores de saltos, em que cada vez que um roteador x não estiver disponível para passar adiante uma mensagem M para um roteador adjacente y (por exemplo, porque a conexão (x,y) caiu), então x armazena a mensagem em uma fila. Seu algoritmo deve trabalhar com uma inicialização dinâmica, em que as conexões de comunicação podem falhar e ser recuperadas. Projete seu protocolo de maneira que, quando uma conexão (x,y) for recuperada, x envia para y todas as mensagens que foram enfileiradas para esta conexão. Mostre como isso permite que as mensagens de inundação sejam enviadas de forma robusta, mesmo na presença de partições temporárias na rede. Quanto espaço adicional é necessário para implementar essa solução? Como esse algoritmo precisa ser modificado para levar em conta computadores que estão permanentemente desconectados da rede?

C-11.11 Faça G ser uma rede representada como um grafo, cujos n nodos representam roteadores e cujas m arestas representam conexões. Suponha também que G é estático (isto é, não muda) e que existe uma árvore de cobertura T definida nos nodos de G. T não necessita ser uma árvore de cobertura mínima, mas inclui todos os nodos de G e, para cada nodo v de G, v armazena seus vizinhos de G a também conhece quais destes são também vizinhos em T. Descreva como utilizar a árvore T para melhorar a complexidade de mensagens na execução do algoritmo de menor caminho *link-state* (que constrói tabelas de roteamento para as rotas de menor caminho). Qual a complexidade de mensagens para este algoritmo revisado?

C-11.12 Suponha que um algoritmo de roteamento, tal como o *link-state* ou o do vetor de distâncias tenha completado sua fase de inicialização em uma rede estável (sem alterações nas ligações). Ou seja, cada roteador i armazena um vetor D, de forma que $D[j]$ armazene a distância do roteador i para o roteador j com o nome do próximo salto no caminho para j. Projete um algoritmo eficiente que permita que todos os roteadores verifiquem que todas as suas tabelas de roteamento estão corretas. Qual a complexidade de mensagens para este algoritmo?

C-11.13 Suponha que estamos implementando *multicasting* usando a abordagem da árvore centrada, de forma que cada membro do grupo de *multicast* dinamicamente entre no grupo enviando uma mensagem de junção em direção à raiz da árvore, e que o caminho percorrido por esta mensagem forma um novo caminho na árvore centrada (unindo-se a algum nodo anterior da árvore centrada). Os membros deixam a árvore de uma forma similar, mas reversa. Descreva como modificar o algoritmo de junção de maneira que, a cada vez que algum roteador x desejar se juntar ao grupo, automaticamente se conecta ao nodo da árvore centrada que está mais próximo de x (observe que esse nodo pode não ser um membro do grupo).

C-11.14 Mostre como estender sua solução do problema anterior de maneira que em vez de encontrar o nodo mais próximo ao roteador x, ele encontre o membro do grupo mais próximo de x. Esse nodo é o candidato ideal para qualquer tipo de requisição que requeira que qualquer membro de um certo grupo responda a uma certa questão.

C-11.15 Suponha que usemos assinaturas digitais no algoritmo da inundação padrão (usando seqüências de números), de maneira que cada roteador que deseja enviar uma mensagem de inundação deva assinar a mensagem. Durante a transmissão, se qualquer roteador receber um pacote que não esta corretamente assinado por seu roteador de origem, então esse pacote é descartado. Este algoritmo permite também que uma fonte x sinalize duas mensagens de inundação diferentes com o mesmo número de seqüência e as envie para vizinhos diferentes ao mesmo tempo. Descreva um algoritmo para detectar se este problema ocorre, assumindo que a rede é duplamente conectada e que x é o único roteador com defeito. Qual a complexidade de mensagens deste algoritmo?

Projetos

P-11.1 Use um protocolo de acesso a rede, tal como "ping" ou "HTTP", para coletar as seguintes estatísticas em quatro hopedeiros da Internet que estejam fisicamente em quatro continentes diferentes (isto é, quatro dos seguintes: América do Norte, América do Sul, Europa, Ásia, África, Austrália e Antártica):

- O mínimo, a média, o desvio-padrão e o máximo do tempo de resposta de 50 solicitações de pacotes consecutivas (todas a poucos segundos umas das outras). Plote seus dados em um gráfico, com o eixo x representando as solicitações e o eixo y representando o tempo de resposta.
- A média do tempo de resposta, medido em 10 solicitações consecutivas de pacotes (todas com alguns segundos entre elas), porém repetidas a cada quatro horas durante pelo menos dois dias. Idealmente, você deve repetir seu experimento de 10 "pings" a cada hora por 5 dias (usando um processo *background*). Plote seus dados com os dias da semana no eixo x e o tempo de resposta no eixo y e sobreponha as estatísticas de todos os hospedeiros usando linhas diferentes no mesmo gráfico.

P-11.2 Faça uma comparação experimental do algoritmo de aproximação da abordagem da árvore de Steiner para *multicast* com a abordagem das árvores centradas. O uso das arestas é significativamente melhor em uma dessas abordagens? Em caso positivo, quais as premissas sobre a distribuição das requisições de junção?

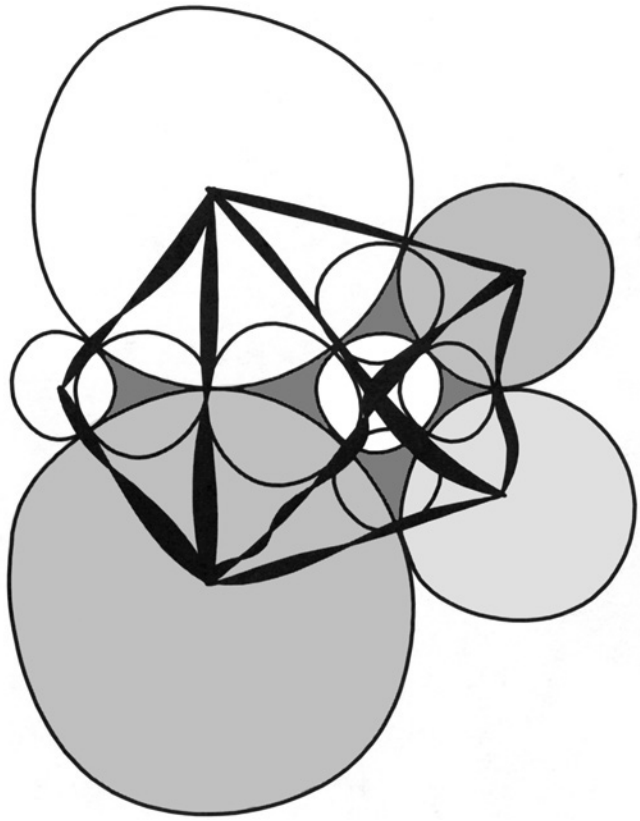
Notas

Para o leitor interessado em estudar mais sobre algoritmos distribuídos, indica-se os excelentes livros de Lynch [137], Peleg [166] e Tel [202]. Para o leitor interessado em leituras adicionais sobre redes de computadores e protocolos, indicam-se os livros de Comer [52], Huitema [105] e Tanenbaum [196]. A implementação distribuída do algoritmo de Barůvka é atribuída a Galleger et al. [74]. Uma forma simples de implementá-lo em uma rede assíncrona é atribuída a Awerbuch [18].

Parte

Tópicos adicionais

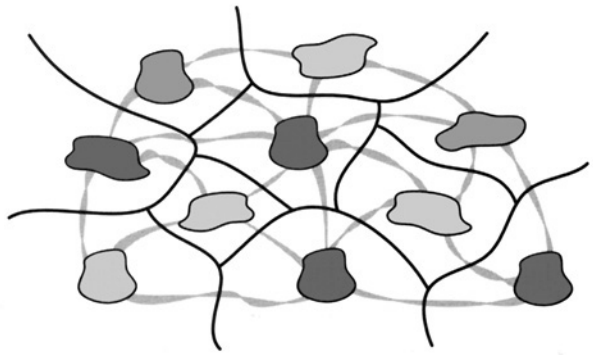
IV



Capítulo

Geometria computacional

12



12.1 Árvores intervalares 540

12.1.1 Pesquisa intervalar em uma dimensão. 541

12.1.2 Pesquisa intervalar bidimensional 544

12.2 Árvores de pesquisa por prioridade. 547

12.2.1 Construção de uma árvore de pesquisa por prioridade 548

12.2.2 Pesquisa em uma árvore de pesquisa por prioridade 549

12.2.3 Árvores intervalares por prioridade 551

12.3 *Quadtrees* e árvores *k*-D 552

12.3.1 *Quadtrees* 552

12.3.2 Árvores *k*-D 554

12.4 A técnica do plano rotacional 556

12.4.1 Intersecção de segmentos ortogonais 556

12.4.2 Determinação do par de pontos mais próximo 559

12.5 Cobertura convexa. 562

12.5.1 Representação de objetos geométricos 563

12.5.2 Teste da orientação dos pontos 564

12.5.3 Propriedades básicas de coberturas convexas. 566

12.5.4 O algoritmo do embrulho para presente 568

12.5.5 O algoritmo de varredura de Graham. 570

12.6 Exemplo em Java: cobertura convexa 573

12.7 Exercícios 577

Vivemos em um mundo geométrico multidimensional. O espaço próprio físico é tridimensional e, por essa razão, usamos três coordenadas, x, y e z, para descrever pontos no espaço. Para descrevermos completamente a orientação da extremidade de um braço de robô, precisamos, na verdade, de seis dimensões: três para descrever a posição da extremidade no espaço mais três para descrever os ângulos em que o braço se encontra (que são tipicamente chamados de *pitch*, *roll* e *yaw**). Descrever a situação de um avião em vôo requer pelo menos nove dimensões: seis para descrever sua orientação da mesma forma que para o braço do robô e mais três para descrever a velocidade do avião. Na verdade, essas representações físicas são consideradas "pouco dimensionais", principalmente frente às aplicações em aprendizado de máquina ou biologia computacional em que espaços com cem e mil dimensões não são incomuns. Este capítulo é dedicado à ***geometria computacional***, que estuda as estruturas de dados e algoritmos para lidar com dados geométricos, tais como pontos, linhas e polígonos.

Existe, na verdade, uma grande variedade de estruturas de dados e algoritmos para processar dados multidimensionais, e está além do escopo deste capítulo discuti-los todos. Em vez disso, porém, faremos uma introdução a alguns dos mais interessantes. Iniciaremos com uma discussão sobre ***árvores intervalares***, que podem armazenar pontos multidimensionais de maneira a auxiliar um tipo especial de operação de pesquisa, chamada de ***consulta intervalar***; também incluímos uma variação interessante da árvore intervalar chamada de ***árvore de prioridades***, que divide o espaço em células; focamos, ainda, variantes conhecidas como ***quadtrees*** e **árvores *k-D*.**

Continuamos a discussão introduzindo um padrão de propósitos gerais chamado de técnica do ***plano rotacional***, que é particularmente útil para resolver problemas geométricos em duas dimensões. Demonstramos a aplicabilidade dessa técnica mostrando como a mesma pode ser usada para resolver problemas de geometria computacional em duas dimensões pela redução dos mesmos em uma série de problemas de uma dimensão. Mais especificamente, aplicamos a técnica do plano rotacional no cálculo da intersecção de um conjunto de segmentos de reta ortogonais e na identificação do par de pontos com distância mínima dentre um conjunto de pontos. A técnica do plano rotacional pode ser usada em muitos outros problemas, alguns dos quais explorados em exercícios ao final do capítulo.

Concluímos este capítulo estudando um problema que tem aplicação em computação gráfica, estatística, sistemas de informação geográficas, robótica e visão computacional. Neste problema, conhecido com o problema da ***cobertura convexa***, estamos interessados em encontrar o menor conjunto convexo que contém um dado conjunto de pontos. Construí-lo nos permitirá definir o "limite" de um conjunto de pontos; logo, este problema levanta algumas considerações interessantes sobre como representar objetos geométricos e executar testes geométricos.

12.1 Árvores intervalares

Dados multidimensionais são usados em uma grande variedade de aplicações, incluindo estatística e robótica. O tipo mais simples de dados multidimensionais são pontos d-dimensionais, que podem ser representados por uma seqüência (x_0, x_1, x_{d-1}) de ***coordenadas*** numéricas. Em aplicações comerciais, um ponto d-dimensional pode repre-

* N. de T. *Pitch*, *roll* e *yaw* são termos técnicos da aviação que designam, respectivamente, levantar ou abaixar o nariz da aeronave, inclinar a aeronave fazendo uma asa subir e outra descer e mudar a direção da aeronave para a esquerda ou para a direita.

sentar os vários atributos de um produto ou pode ser empregado em um banco de dados. Por exemplo, televisões em um catálogo eletrônico têm provavelmente diferentes valores de atributos para preço, tamanho da tela, peso, largura e profundidade. Os dados multidimensionais podem também ser originários de aplicações científicas, em que cada ponto representa atributos de experimentos ou observações individuais. Por exemplo, objetos celestes em uma varredura astronômica do céu provavelmente terão diferentes valores para brilho (ou magnitude aparente), diâmetro, distância e posição no céu (que é também bidimensional).

Uma consulta normal a ser feita em um conjunto multidimensional de pontos é a ***consulta intervalar***, que é uma solicitação para que sejam recuperados todos os pontos de uma coleção multidimensional cujas coordenadas estejam dentro de certos limites. Por exemplo, um consumidor que deseja comprar uma televisão nova pode solicitar a partir do catálogo eletrônico de uma loja, todas as unidades que tenham tamanho de tela entre 24 e 27 polegadas e preço entre $ 200 e $ 400. Em outro caso, um astrônomo interessado em estudar asteróides pode solicitar todos os objetos celestes que estejam a uma distância entre 1,5 e 10 unidades astronômicas, tenham magnitude aparente entre +1 e +15 e diâmetro entre 0,5 e mil quilômetros. A estrutura de dados de árvore intervalar, que discutimos nesta seção, pode ser usada para responder a essas questões.

Pesquisa intervalar em duas dimensões

Para simplificar a discussão, vamos focar as consultas intervalares de duas dimensões. O Exercício C-12.6 referencia a forma como as estruturas bidimensionais podem ser estendidas para dimensões maiores. Um ***dicionário bidimensional*** é um TAD para armazenar itens elemento-chave cuja chave é um par (x,y) de números, chamados de coordenadas do elemento. Um dicionário bidimensional D suporta a seguinte operação de consulta:

findAllInRange(x_1, y_2, y_1, y_2): retorna todos os elementos de D com coordenadas (x,y) tal que $x_1 \leq x \leq x_2$ e $y_1 \leq y \leq y_2$.

A operação finAllInRange é a versão ***relatório*** da consulta intervalar, porque pergunta por todos os itens que satisfaçam às restrições intervalares. Existe também uma versão ***contadora*** da consulta intervalar, na qual estamos interessados simplesmente na quantidade de itens no intervalo. Apresentamos as estruturas de dados para construir um dicionário bidimensional no restante desta seção.

12.1.1 Pesquisa intervalar em uma dimensão

Antes de passarmos para a pesquisa intervalar em duas dimensões, faremos uma pequena digressão para estudar a ***pesquisa intervalar em uma dimensão***, onde, dado um dicionário ordenado D, queremos executar a seguinte operação de consulta:

findAllInRange(k_1,k_2): retorna todos os elementos do dicionário D com chave k onde $k_1 \leq k \leq k_2$.

Vamos discutir como podemos usar uma árvore de pesquisa binária T que representa o dicionário D (veja o Capítulo 3) para executar uma consulta findAllInRange(k_1,k_2). Usaremos um método recursivo chamado de 1DtreeRangeSearch que recebe como argumentos os parâmetros intervalares k_1 e k_2 e um nodo v de T. Se o nodo v

for externo, está pronto. Se o nodo v for interno, então temos três casos dependendo do valor de key(v), a chave do item armazenado no nodo v:

- key(v) < k_1: a recursão ocorre sobre o filho da direita de v;
- $k_1 \leq$ key(v) $\leq k_2$: relatamos element(v) e a recursão ocorre em ambos filhos de v;
- key(v) > k_2: a recursão ocorre sobre o filho da esquerda de v.

Descrevemos esse procedimento no Algoritmo 12.1 (1DTreeRangeSearch) e o ilustramos na Figura 12.2. Executamos a operação findAllInRange(k_1,k_2) chamando 1DTreeRangeSearch(k_1,k_2,T.root()). Intuitivamente, o método 1DTreeSearchRange é uma modificação do método padrão de pesquisa em árvore binária (Algoritmo 3.5), permitindo pesquisar "ambas"as chaves k_1 e k_2.

Algoritmo 1DTreeRangeSearch(k_1,k_2,v):
  Entrada: chaves de busca k_1 e k_2 e um nodo v pertencente a uma árvore binária T.
  Saída: os elementos armazenados na subárvore de T enraizada em v, cujas chaves são maiores ou iguais a k_1 e menores ou iguais a k_2.
  se T.isExternal(v) **então**
    retorne ∅
  se $k_1 \leq$ key(v) $\leq k_2$ **então**
    $L \leftarrow$ 1DTreeRangeSearch(k_1,k_2,T.leftChild(v))
    $R \leftarrow$ 1DTreeRangeSearch(k_1,k_2,T.rightChild(v))
    retorne $L \cup \{$element(v)$\} \cup R$
  senão se key(v) < k_1 **então**
    retorne 1DTreeRangeSearch(k_1,k_2,T.rightChild(v))
  senão se k_2 < key(v) **então**
    retorne 1DTreeRangeSearch(k_1,k_2,T.leftChild(v))

Algoritmo 12.1 Método recursivo para pesquisa intervalar unidimensional em uma árvore de pesquisa binária.

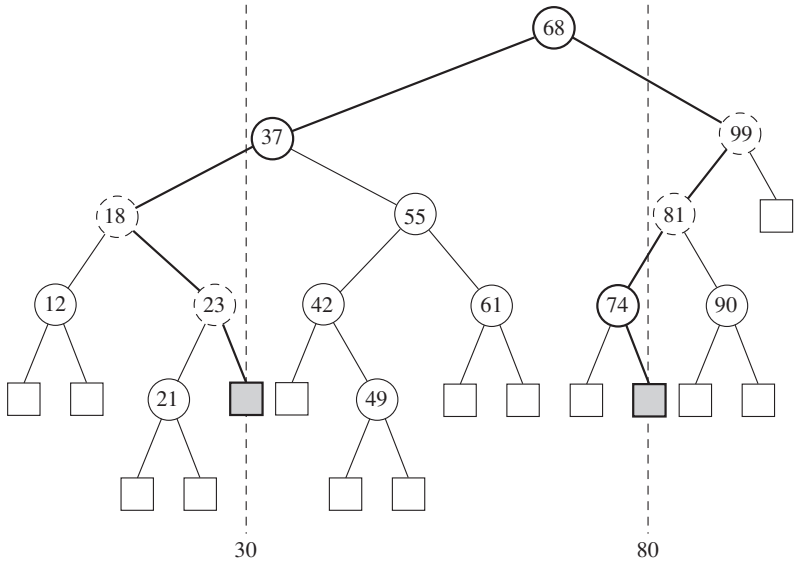


Figura 12.2 Pesquisa intervalar unidimensional usando uma árvore de pesquisa binária para $k_1 = 30$ e $k_2 = 80$. Os caminhos P_1 e P_2 dos nodos do limite são desenhados com linhas grossas. Os nodos do limite que armazenam itens com chave fora do intervalo $[k_1,k_2]$ são desenhados com linhas tracejadas. Existem 10 nodos internos.

Desempenho

Analisaremos agora o tempo de execução do algoritmo 1DTreeRangeSearch. Em nossa análise, assumiremos por simplicidade que T não contém itens com chaves k_1 ou k_2. A extensão dessa análise para o caso geral é deixada como exercício.

Faça P_1 ser o caminho percorrido quando executando uma pesquisa na árvore T pela chave k_1. O caminho P_1 inicia na raiz de T e termina em um nodo externo de T. Definamos um caminho similar, P_2, em relação a k_2. Identificamos cada nodo v de T como pertencente a um dos três grupos (veja a Figura 12.2):

- O nodo v é um ***nodo limite*** se v pertencer a P_1 ou P_2; um nodo limite armazena um item cuja chave pode estar dentro ou fora do intervalo $[k_1,k_2]$.
- O nodo v é um ***nodo interno*** se v não for um nodo limite e v pertencer a uma subárvore enraizada no filho direito de um nodo de P_1 ou ao filho esquerdo de um nodo de P_2; um nodo interno armazena um item cuja chave pertence ao intervalo $[k_1,k_2]$.
- O nodo v é um ***nodo externo*** se v não for um nodo limite e v pertencer a subárvore enraizada no filho esquerdo de um nodo de P_1 ou no filho direito de um nodo de P_2; um nodo externo armazena um item cuja chave está fora do intervalo $[k_1,k_2]$.

Consideremos a execução do algoritmo 1DTreeRangeSearch(k_1,k_2,r), onde r é a raiz de T. Percorremos um caminho composto por nodos limite chamando o algoritmo recursivamente, tanto sobre o filho da esquerda como sobre o filho da direita, até atingirmos um nodo externo ou um nodo interno w (que pode ser a raiz) com chave no intervalo $[k_1,k_2]$. No primeiro caso (atingimos um nodo externo), o algoritmo se encerra retornando um conjunto vazio. No segundo caso, a execução continua ativando o algoritmo recursivamente sobre os dois filhos de w. Sabemos que o nodo w é o nodo mais baixo dos nodos comuns aos caminhos P_1 e P_2. Para cada nodo limite v visitado a partir deste ponto, ou faremos uma chamada única no filho de v, que é também um nodo limite, ou faremos uma chamada em um filho de v que é nodo limite e no outro filho que é um nodo interno. Uma vez que visitamos um nodo interno, iremos visitar todos os seus descendentes (de nodos internos).

Uma vez que gastamos uma quantidade constante de trabalho em cada nodo visitado pelo algoritmo, o tempo de execução do algoritmo é proporcional ao número de nodos visitados. Contamos os nodos visitados como segue:

- Não visitamos nodos externos.
- Visitamos no máximo $2h + 1$ nodos limite, onde h é a altura de T, uma vez que os nodos limites estão nos caminhos de pesquisa P_1 e P_2 e compartilham pelo menos um nodo (o nodo raiz T).
- Cada vez que visitamos um nodo interno v, também visitamos toda a subárvore T_v de T enraizada em v, e adicionamos todos os elementos armazenados nos nodos internos de T_v no conjunto relatado. Se T_v contém s_v itens, então ele tem $2s_v + 1$ nodos. Os nodos internos podem ser divididos em j subárvores disjuntas $T_1,\ldots, T_j$ enraizadas nos filhos dos nodos limites, onde $j \leq 2h$. Denotando

com s_i o número de itens armazenados na árvore T_i, temos que o número total de nodos internos visitados é

$$\sum_{i=1}^{j}(2s_i + 1) = 2s + j \leq 2s + 2h.$$

Por essa razão, no máximo $2s + 4h + 1$ nodos de T são visitados, e a operação findAllInRange executa em tempo $O(h + s)$. Se quisermos minimizar h no pior caso, podemos usar uma árvore binária de pesquisa balanceada para T, tal como uma árvore AVL (Seção 3.2) ou uma árvore vermelho-preta (Seção 3.3), de maneira que h seja $O(\log n)$. Além disso, usando uma árvore binária de pesquisa balanceada, podemos executar também as operações insertItem e removeElement em tempo $O(\log n)$ cada. Resumindo:

Teorema 12.1: *uma árvore balanceada de pesquisa suporta pesquisa intervalar unidimensional em um dicionário ordenado com n itens:*

- *O espaço usado é* $O(n)$.
- *A operação* findAllInRange *consome tempo* $O(\log n + s)$, *onde s é o número de elementos relatados.*
- *As operações* insertItem *e* removeElement *consomem tempo* $O(\log n)$ *cada.*

12.1.2 Pesquisa intervalar bidimensional

A árvore de pesquisa intervalar bidimensional (Figura 12.3) é uma estrutura de dados que implementa o TAD dicionário bidimensional. Ela consiste em uma estrutura ***primária***, que é uma árvore de pesquisa binária balanceada T, com um conjunto de estruturas ***auxiliares***. Mais especificamente, como descreveremos a seguir, cada nodo interno da estrutura primária T armazena uma referência para uma estrutura auxiliar relacionada. A função da estrutura primária, T, é suportar pesquisa baseada em abscissas (coordenadas x). Para suportar também a pesquisa em função de ordenadas (coordenadas y), usamos uma coleção de estruturas de dados auxiliares, cada uma correspondendo a uma árvore de pesquisa intervalar unidimensional que usa ordenadas como chave. A estrutura primária de T é uma árvore de pesquisa binária balanceada construída usando as abscissas dos itens como chaves. Um nodo interno v de T armazena os seguintes dados:

- um item, cujas coordenadas são denotadas por $x(v)$ e $y(v)$, e cujo elemento é denotado por element(v);
- uma árvore intervalar unidimensional $T(v)$ que armazena o mesmo conjunto de itens como subárvore enraizada em v de T (incluindo v), mas usando as ordenadas como chaves.

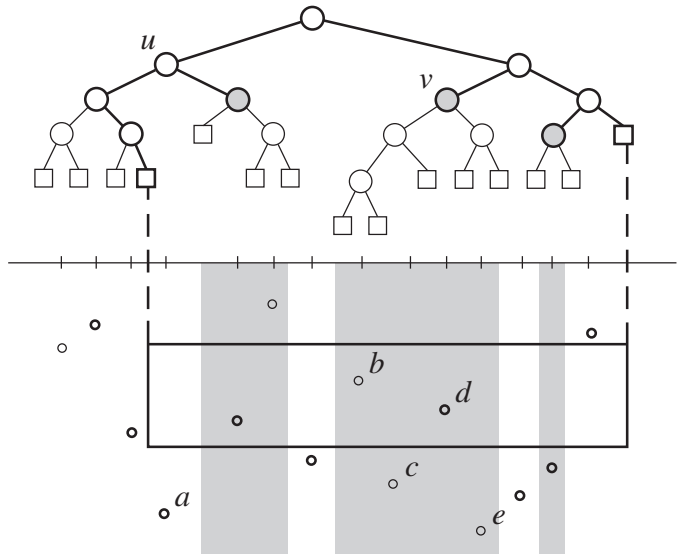


Figura 12.3 Um conjunto de itens com chaves bidimensionais representadas por uma árvore intervalar bidimensional e uma pesquisa intervalar sobre a mesma. A estrutura primária T é apresentada. Os nodos de T visitados pelo algoritmo de pesquisa são desenhados com linhas grossas. Os nodos limites são preenchidos com branco, e os nodos alocados são preenchidos com cinza. O ponto a, armazenado no nodo limite u, está fora do intervalo de pesquisa. As listas verticais cinza cobrem os pontos armazenados nas estruturas auxiliares dos nodos alocados. Por exemplo, a estrutura auxiliar do nodo v armazena os pontos b, c, d e e.

Lema 12.2: *uma árvore intervalar bidimensional armazenando n itens ocupa espaço* $O(n \log n)$ *e pode ser construída em tempo* $O(n \log n)$.

Prova: a estrutura primária usa espaço $O(n)$. Existem n estruturas secundárias. O tamanho de uma estrutura auxiliar é proporcional ao número de itens armazenados na mesma. Um item armazenado em um nodo v da estrutura primária T é também armazenado em cada estrutura auxiliar $T(u)$, uma vez que u é um ancestral de v. Como a árvore T é balanceada, o nodo v tem $O(\log n)$ ancestrais. Assim, existem $O(\log n)$ cópias do item na estrutura auxiliar. Logo, o espaço total utilizado é $O(n \log n)$. A construção do algoritmo é deixada como exercício (C-12.2). ■

O algoritmo da operação findAllInRange(x_1,x_2,y_1,y_2) inicia executando o que é essencialmente uma pesquisa intervalar unidimensional na estrutura primária T para o intervalo $[x_1,x_2]$. Na verdade, percorremos a árvore T para baixo procurando por nodos internos. Entretanto, fazemos uma modificação importante: quando encontramos um nodo interno v, em vez de visitarmos recursivamente a subárvore enraizada em v, executamos uma pesquisa intervalar unidimensional para o intervalo $[y_1,y_2]$ na estrutura auxiliar de v.

Chamamos de ***nodos de alocação*** os nodos internos de T que são filhos dos nodos limites. O algoritmo visita os nodos limites e os nodos de alocação de T, mas não os outros nodos internos. Cada nodo limite v é classificado pelo algoritmo como um ***nodo esquerdo***, ***nodo central*** ou ***nodo direito***. Um nodo central está na intersecção dos caminhos de pesquisa P_1 para x_1 e P_2 para x_2. Um nodo esquerdo está em P_1, mas não em P_2. Um nodo direito esta em P_2, mas não em P_1. Em cada nodo de alocação v, o algoritmo executa uma pesquisa intervalar unidimensional na estrutura auxiliar $T(v)$ pa-

ra o intervalo das ordenadas $[y_1,y_2]$. Fornecemos os detalhes desse método no Algoritmo 12.4 (veja também a Figura 12.3).

Teorema 12.3: *uma árvore intervalar bidimensional T para um conjunto de n itens com chaves bidimensionais usa espaço $O(n \log n)$ e pode ser construída em tempo $O(n \log n)$. Usando T, uma consulta intervalar bidimensional consome tempo $O(\log^2 n + s)$, onde s é o número de elementos relatados.*

Prova: os requisitos de espaço e tempo de construção são originários do Lema 12.2. Analisaremos agora o tempo de execução de uma consulta intervalar executada com o Algoritmo 12.4 (2DTreeRangeSearch). Contabilizamos o tempo gasto em cada nodo limite e de alocação da estrutura primária T. O algoritmo consome um tempo constante em cada nodo limite. Uma vez que existem $O(\log n)$ nodos limites, o tempo total gasto nos nodos limites é $O(\log n)$. Para cada nodo de alocação v, o algoritmo gasta tempo $O(\log n_v + s_v)$ executando uma pesquisa intervalar unidimensional na estrutura auxiliar $T(v)$, onde n_v é o número de itens armazenados em $T(v)$. Denotando com A o conjunto de nodos de alocação, temos que o tempo total gasto nos nodos de alocação é proporcional a $\Sigma_{v \in A}(\log n_v + s_v)$. Uma vez que $|A|$ é $O(\log n)$, $n_v \leq n$ e $\Sigma_{v \in A} \leq s$, temos que o tempo total gasto nos nodos de alocação é $O(\log^2 n + s)$. Concluímos que uma pesquisa operacional bidimensional consome tempo $O(\log^2 n + s)$. ■

Algoritmo 2DTreeRangeSearch(x_1, x_2, y_1, y_2, v, t):
  Entrada: chaves de pesquisa x_1, x_2, y_1 e y_2; nodo v na estrutura primária T de uma árvore intervalar bidimensional; tipo t do nodo v.
  Saída: os itens na subárvore enraizada em v cujas coordenadas estão no intervalo das abscissas $[x_1, x_2]$ e no intervalo das ordenadas $[y_1,y_2]$
  se T.isExternal(v) **então**
    retorne $\emptyset$
  se $x_1 \leq x(v) \leq x_2$ **então**
    se $y_1 \leq y(v) \leq y_2$ **então**
      $M \leftarrow \{$element(v)$\}$
    senão
      $M \leftarrow \emptyset$
    se t = "left" **então**
      $L \leftarrow$ 2DTreeRangeSearch(x_1,x_2,y_1,y_2,T.leftChild(v),"left")
      $R \leftarrow$ 1DTreeRangeSearch(y_1,y_2,T.rightChild(v))
    senão se t = "right" **então**
      $L \leftarrow$ 1DTreeRangeSearch(y_1,y_2,T.leftChild(v))
      $R \leftarrow$ 2DTreeRangeSearch(x_1,x_2,y_1,y_2,T.rightChild(v), "right")
    senão
      $\{t$ = "central"$\}$
      $L \leftarrow$ 2DTreeRangeSearch(x_1,x_2,y_1,y_2,T.leftChild(v),"left")
      $R \leftarrow$ 2DTreeRangeSearch(x_1,x_2,y_1,y_2,T.rightChild(v), "right")
  senão
    $M \leftarrow \emptyset$
    se $x(v) < x_1$ **então**

$L \leftarrow \emptyset$
$R \leftarrow$ 2DTreeRangeSearch(x_1,x_2,y_1,y_2,T.rightChild(v), t)
senão
$\{x(v) > x_2\}$
$L \leftarrow$ 2DTreeRangeSearch(x_1,x_2,y_1,y_2,T.leftChild(v),t)
$R \leftarrow \emptyset$
retorne $L \cup M \cup R$

Algoritmo 12.4 Método recursivo para pesquisa intervalar bidimensional em uma árvore de pesquisa intervalar bidimensional. A chamada inicial do método é 2DTreeRangeSearch (x_1,x_2,y_1,y_2,T.root(),"central"). O algoritmo é chamado recursivamente em todos os nodos limite em relação ao intervalo das abscissas $[x_1, x_2]$. O parâmetro t indica quando v é um nodo limite esquerdo, direito ou central.

12.2 Árvores de pesquisa por prioridade

Nesta seção, apresentamos a estrutura ***árvore de pesquisa por prioridade*** que pode responder a consultas intervalares de ***três lados*** sobre um conjunto S de itens com chaves bidimensionais:

findAllInRange(x_1, x_2, y_1): retorna todos os itens de S com coordenadas (x,y) tais que $x_1 \leq x \leq x_2$ e $y_1 \leq y$.

Geometricamente, essa consulta retorna todos os pontos entre duas linhas verticais ($x = x_1$ e $x = x_2$) e sobre uma linha horizontal ($y = y_1$).

Uma ***árvore de pesquisa por prioridade*** para um conjunto S é uma árvore binária que armazena os itens de S que se comportam como uma árvore binária, no que diz respeito às abscissas, e como um *heap*, no que diz respeito às ordenadas. Por simplicidade, vamos assumir que todos itens de S têm coordenadas x e y distintas. Se o conjunto S estiver vazio, T consiste em um único nodo externo. Em qualquer outro caso, faça $\bar{p}$ ser o item mais alto de S, ou seja, o item com a ordenada máxima. Denotamos com $\hat{x}$ a abscissa mediana dos itens em $S - \{\bar{p}\}$, e com S_L e S_R os subconjuntos de $S - \{\bar{p}\}$ com itens tendo abscissas menores ou iguais a $\hat{x}$ e maiores que $\hat{x}$, respectivamente. Recursivamente definimos a árvore de pesquisa por prioridade, T para S, como segue:

- A raiz T armazena o item $\bar{p}$ e a abscissa mediana $\hat{x}$.
- A subárvore esquerda de T é uma árvore de pesquisa por prioridade para S_L.
- A subárvore direita de T é uma árvore de pesquisa por prioridade para S_R.

Para cada nodo interno v de T, denotamos com $\bar{p}(v)$, $\hat{x}(v)$ e $\bar{y}(v)$ o item mais alto armazenado em v e suas coordenadas. Também, denotamos com $\hat{x}(v)$ a abscissa mediana armazenada em v. Um exemplo de árvore de pesquisa por prioridade é mostrado na Figura 12.5.

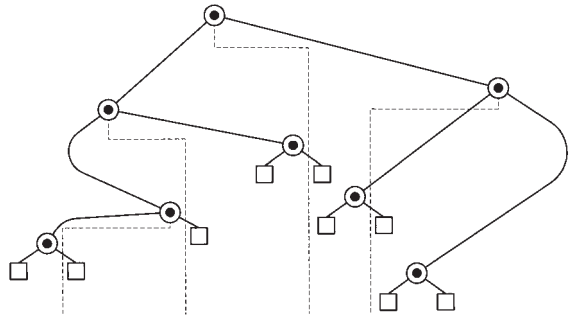


Figura 12.5 Um conjunto S de itens com chaves bidimensionais e uma árvore de pesquisa por prioridades T para S. Cada nodo interno v de T é desenhado como um círculo ao redor de $\bar{p}\ (v)$. A abscissa mediana $\hat{x}(v)$ é representada por uma linha tracejada abaixo do nodo v que separa os itens armazenados na subárvore esquerda de v daqueles armazenados na subárvore direita.

12.2.1 Construção de uma árvore de pesquisa por prioridades

As ordenadas dos itens armazenados nos nodos de uma árvore de pesquisa por prioridade T satisfazem a propriedade de ordem do *heap* (Seção 2.4.3). Isto é, se u é o pai de v, então $\bar{y}(u) > \bar{y}(v)$. Também as abscissas medianas armazenadas nos nodos de T definem uma árvore de pesquisa binária (Seção 3.1.2). Esses dois fatos motivam o termo "árvore de pesquisa por prioridades". Vamos, entretanto, explicar como construir uma árvore de pesquisa por prioridade a partir de um conjunto S de n itens bidimensionais. Começamos ordenando S em ordem crescente de abscissa e então chamamos o método recursivo buildPST(S) apresentado no Algoritmo 12.6.

Algoritmo buildPST(S):
  Entrada: uma seqüência S de n itens bidimensionais ordenados pela abscissa.
  Saída: uma árvore de pesquisa por prioridades T para S.
  Criar uma árvore binária elementar T consistindo em um único nodo externo v
  se !S.isEmpty() **então**
    Percorra a seqüência S para encontrar o item $\bar{p}$ de S com maior ordenada
    Remova $\bar{p}$ de S
    $\bar{p}\ (v) \leftarrow \bar{p}$
    $\hat{p} \leftarrow S$.elemAtRank($\lceil S$.size()/2$\rceil$)
    $\hat{x}(v) \leftarrow x(\hat{p})$
    Divida S em duas subseqüências, S_L e S_R, onde S_L contém os items acima de $\hat{p}$ (inclusive) e S_R contém os itens restantes
    $T_L \leftarrow$ buildPST(S_L)
    $T_R \leftarrow$ buildPST(S_R)
    T.expandExternal(v)
    Substitui o filho da direita de v por T_L
    Substitui o filho da esquerda de v por T_R
  retorne T

Algoritmo 12.6 Construção recursiva de uma árvore de pesquisa por prioridade.

Lema 12.4: *dado um conjunto S de n itens bidimensionais, uma árvore de pesquisa por prioridade S usa espaço $O(n)$, tem altura $O(\log n)$ e pode ser construída em tempo $O(n \log n)$.*

Prova: o espaço $O(n)$ requerido resulta do fato de que todo nodo interno da árvore de pesquisa por prioridade T armazena um item distinto de S. A altura de T resulta da divisão em partes iguais do número de nodos em cada nível. A ordenação preliminar dos itens de S pelas abscissas pode ser feita em tempo $O(n \log n)$ usando um algoritmo de ordenação assintóticamente ótimo, tal como o *heapsort* ou o *merge-sort*. O tempo de execução $T(n)$ do método buildPST (Algoritmo 12.6) caracteriza-se pela recorrência $T(n) = 2T(n/2) + bn$, para uma constante $b > 0$. Entretando, pelo Teorema Mestre (5.6), $T(n)$ é $O(n \log n)$. ■

12.2.2 Pesquisa em uma árvore de pesquisa por prioridade

Mostraremos agora como executar uma consulta com uma variação trilateral FindAllInRange(x_1,x_2,x_3) em uma árvore de pesquisa por prioridade T. Percorremos T para baixo de uma forma similar à da pesquisa intervalar unidimensional para o intervalo $[x_1,x_2]$. Uma diferença importante, entretanto, é que continuamos a pesquisa em uma subárvore de um nodo v se $y(v) \geq y_1$. Fornecemos os detalhes do algoritmo para a pesquisa trilateral no Algoritmo 12.7 (PSTSearch) e demonstramos a sua execução na Figura 12.8.

Algoritmo PSTSearch(x_1, x_2, y_1, v):
  Entrada: intervalo de três lados, definido por x_1, x_2 e y_1, e o nodo v de uma árvore de pesquisa intervalar por prioridade T.
  Saída: Os itens armazenados na subárvore enraizada em v com coordenadas (x,y) tais que $x_1 \leq x_2$ e $y_1 \leq y$.
  se $\bar{y}(v) < y_1$ **então**
    retorne $\emptyset$
  se $x_1 \leq \bar{x}(v) \leq x_2$ **então**
    $M \leftarrow \{\bar{p}(v)\}$ {devemos exibir $\bar{p}(v)$}
  senão
    $M \leftarrow \emptyset$
  se $x_1 \leq \hat{x}(v)$ **então**
    $L \leftarrow$ PSTSearch(x_1, x_2,y_1,T.leftChild(v))
  senão
    $L \leftarrow \emptyset$
  se $\hat{x}(v) \leq x_2$ **então**
    $R \leftarrow$ PSTSearch(x_1, x_2,y_1,T.rightChild(v))
  senão
    $R \leftarrow \emptyset$
  retorne $L \cup M \cup R$

Algoritmo 12.7 Pesquisa intervalar de três lados em uma árvore de pesquisa por prioridade T. O algoritmo é chamado inicialmente com PSTSearch(x_1, x_2,y_1,T.root()).

Observamos que definimos intervalos de três lados para ter os lados da direita, da esquerda e de baixo, e para não estar limitado acima. Essa restrição foi feita, entretanto, sem perda de generalidade, pois podemos definir nossas consultas trilaterais usando

qualquer um dos três lados de um retângulo. A árvore de pesquisa por prioridade desta definição alternativa é similar à definida anteriormente, mas "virada nos seus lados".

Vamos analisar o tempo de execução do método PSTSearch para responder a uma consulta trilateral em uma árvore de pesquisa por prioridade T que armazena um conjunto de n itens com chaves bidimensionais. Denotamos com s o número de itens relatados. Uma vez que em cada nodo visitado consumimos tempo $O(1)$, o tempo de execução do método PSTSearch é proporcional ao número de nodos visitados.

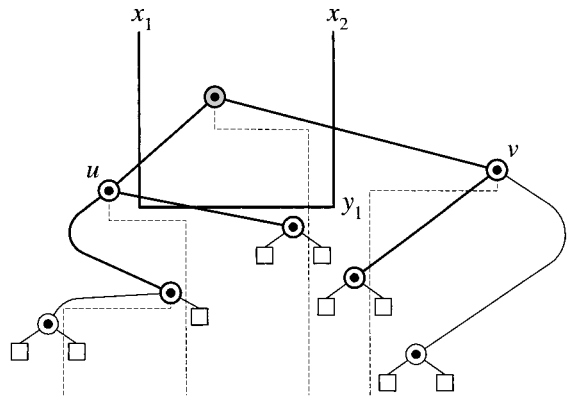


Figura 12.8 Pesquisa intervalar trilateral em uma árvore de pesquisa por prioridade. Os nodos visitados são desenhados com linhas grossas. Os nodos que armazenam os itens relatados são pintados de cinza.

Cada nodo v visitado pelo método PSTSearch é classificado como segue:

- O nodo v é um ***nodo limite*** se estiver no caminho de pesquisa de x_1 ou x_2 quando enxergando T como uma árvore de pesquisa binária sobre as abscissas medianas armazenadas em seus nodos. O item armazenado em um nodo interno limite pode estar dentro ou fora do intervalo de três lados. Pelo Lema 12.4, a altura de T é $O(\log n)$. Assim, existem $O(\log n)$ nodos limite.
- O nodo v é um ***nodo interno*** se for interno, não for um nodo limite e $\bar{y}(v) \geq y_1$. O item armazenado em um nodo interno está dentro do intervalo de três lados. O número de nodos internos não é maior do que o número s de itens relatados.
- O nodo v é um ***nodo terminal*** se não for um nodo limite e, se interno, $\bar{y}(v) < y_1$. O item armazenado em um nodo interno terminal está fora do intervalo de três lados. Cada nodo terminal é filho de um nodo limite ou de um nodo interno. Assim, o número de nodos terminais é no máximo o dobro do número de nodos limite mais os nodos internos. Logo, existem $O(\log n + s)$ nodos terminais.

Concluímos que PSTSearch visita $O(\log n + s)$ nodos, fornecendo-nos o seguinte:

Teorema 12.5: *uma árvore de pesquisa por prioridade T que armazena n itens com chaves bidimensionais usa espaço $O(n)$ e pode ser construída em tempo $O(n \log n)$. Usando T, uma consulta de três lados consome tempo $O(\log n + s)$, onde s é o número de itens relatados.*

Naturalmente, consultas intervalares trilaterais não são tão genéricas como consultas intervalares normais (de quatro lados), que podem ser respondidas em tempo

$O(\log^2 n + k)$ usando a estrutura de dados de árvore intervalar discutida na seção anterior. Porém, árvores de pesquisa por prioridade podem ser usadas para acelerar o tempo de execução de respostas consultas intervalares padrão bidimensionais de quatro lados. A estrutura de dados resultante, conhecida como ***árvore intervalar por prioridades***, usa árvores de pesquisa por prioridade como estruturas auxiliares de uma forma que obtém o mesmo espaço limitado das árvores intervalares tradicionais. Discutimos esta estrutura de dados na seqüência.

12.2.3 Árvores intervalares por prioridade

Faça T ser uma árvore binária balanceada que armazena n itens com chaves bidimensionais, ordenadas de acordo com suas abscissas. Mostraremos como incrementar T com árvores de pesquisa por prioridade como estruturas auxiliares para responder a consultas intervalares (de quatro lados). A estrutura de dados resultante é chamada de ***árvore intervalar por prioridade***.

Para converter T em uma árvore intervalar por prioridade, visitamos todos os nodos internos v de T, com exceção da raiz, e construímos, como estrutura auxiliar, uma árvore de pesquisa por prioridade $T(v)$ para os itens armazenados na subárvore de T enraizada em v. Se v for um filho da esquerda, $T(v)$ responde a consultas sobre intervalos trilaterais sem limite na direita. Se v for um filho da direita, $T(v)$ responde a consultas sobre intervalos trilaterais sem limite na esquerda. Pelos Lemas 12.2 e 12.4, uma árvore intervalar por prioridade usa espaço $O(n \log n)$ e pode ser construída em tempo $O(n \log n)$. O método para executar uma consulta intervalar bidimensional em uma árvore intervalar por prioridade é fornecido no Algoritmo 12.9 (PSTRangeSearch).

Algoritmo PSTRangeSearch(x_1, x_2, y_1, v):
  Entrada: chaves de pesquisa x_1, x_2, y_1 e y_2; nodo v na estrutura primária T de uma árvore intervalar com prioridade.
  Saída: os itens da subárvore enraizada em v cujas coordenadas estão no intervalo das abscissas $[x_1, x_2]$ e no intervalo das ordenadas $[y_1, y_2]$.
  se T.isExternal(v) **então**
    retorne $\emptyset$
  se $x_1 \leq x(v) \leq x_2$ **então**
    se $y_1 \leq y(v) \leq y_2$ **então**
      $M \leftarrow \{$element(v)$\}$
    senão
      $M \leftarrow \emptyset$
    $L \leftarrow$ PSTSearch(x_1,y_1,y_2,T(leftChild(v).root())
    $R \leftarrow$ PSTSearch(x_2,y_1,y_2,T(rigthChild(v).root())
    retorne $L \cup M \cup R$
  senão se $x(v) < x_1$ **então**
    retorne PSTRangeSearch(x_1, x_2, y_1, y_2,T.rigthChild(v))
  senão
    $\{ x_2 < x(v) \}$
    retorne PSTRangeSearch(x_1, x_2, y_1, y_2,T.leftChild(v))

Algoritmo 12.9 Pesquisa intervalar em uma árvore intervalar com prioridade T. O algoritmo é iniciado chamando PSTRangeSearch(x_1, x_2, y_1, y_2,T.root())

Teorema 12.6: *uma árvore intervalar com prioridade T para um conjunto de n itens com chaves bidimensionais usa espaço* $O(n \log n)$ *e pode ser construído em tempo* $O(n \log n)$. *Usando T, uma consulta intervalar bidimensional leva tempo* $O(\log n + s)$, *onde s é o número de elementos relatados.*

12.3 *Quadtrees* e árvores *k*-D

Conjuntos de dados multidimensionais normalmente são gerados por grandes aplicações; por esta razão, preferem-se estruturas de dados cuja ocupação de espaço seja linear para armazená-los. Uma arquitetura genérica para projetar estruturas com consumo linear de espaço para dados d-dimensionais, onde a dimensionalidade d é constante, é baseada em uma abordagem chamada de árvore de partição.

Uma ***árvore de partição*** é uma árvore enraizada T com no máximo n nodos externos, onde n é o número de pontos d-dimensionais em um dados conjunto S. Cada nodo externo da árvore de partição T armazena um pequeno subconjunto de S. Cada nodo interno v em uma árvore de partição corresponde a uma região do espaço d-dimensional, o qual é dividido em uma certa quantidade c de células ou regiões diferentes associadas com os filhos de v. Para cada região R associada com um filho u de v, requeremos que todos os pontos na subárvore u pertençam à região R. No caso ideal, os filhos de v correspondem a c células distintas que podem ser facilmente distinguidas usando um número constante de comparações e cálculos aritméticos.

12.3.1 *Quadtrees*

A primeira estrutura de dados árvore de partição que discutiremos é a ***quadtree***. A principal aplicação de *quadtrees* é em conjuntos de pontos originários de imagens, em que as abscissas e as ordenadas são números inteiros, porque os pontos são originários dos *pixels* de uma imagem. O potencial desta estrutura será melhor explorado se a distribuição dos pontos for ligeiramente não-uniforme, com algumas áreas em grande parte vazias e outras bastante densas.

Suponhamos que temos um conjunto S de n pontos no plano. Além disso, façamos R denotar uma região quadrada que contém todos os pontos de S (por exemplo, R pode ser a caixa limitante de uma imagem de 2048×2048 que produz o conjunto S). A estrutura de dados *quadtree* é uma árvore de partição T cuja raiz r está associada com a região R. Para obter o próximo nível de T, subdividimos R em quatro quadrados de mesmo tamanho, R_1, R_2, R_3 e R_4, e associamos cada quadrado R_i com um filho em potencial da raiz r. Mais especificamente, criamos um filho v_i de r se o quadrado R_i contiver um ponto de S. Se um quadrado R_i não contém pontos de S, então não criamos filho de r para ele. Esse processo de refinar R em quadrados R_1, R_2, R_3 e R_4 é chamado de ***divisão***.

A ***quadtree*** T é definida pela aplicação recursiva da divisão sobre cada filho v de r quando necessário. Isto é, cada filho v de r tem uma região quadrada R_i associada consigo e, se a região R_i de v contiver mais de um ponto de S, então executamos uma divisão em v, subdividindo R_i em quatro quadrados de igual tamanho e representando este processo de subdivisão em v. Continuamos dessa forma, dividindo os quadrados

que contêm mais de um ponto em quatro subquadrados e aplicando recursão nos quadrados não-vazios até ter separados todos os pontos de S em quadrados individuais. Armazenamos, então, cada ponto p de S em um nodo externo de T que corresponde ao menor quadrado no processo de divisão que possa conter p. Armazenamos em cada nodo interno v uma representação concisa da divisão executada para v.

Demonstramos um exemplo de conjunto de pontos e a *quadtree* correspondente na Figura 12.10. Observamos, entretanto, que, contrariando o desenho, não existe potencial limite superior na profundidade da *quadtree*, como definimos anteriormente. Por exemplo, nosso conjunto de pontos S pode conter dois pontos que estão muito próximos um do outro, e pode ser que tenhamos uma seqüência muito grande de divisões antes que consigamos separar um do outro. Logo, é normal que os projetistas de *quadtrees* definam um limite superior D para a profundidade de T. Dado um conjunto S de n pontos no plano, podemos construir uma *quadtree* T para S gastando tempo $O(n)$ em cada nível de T. Assim, no pior caso, construir uma *quadtree* com limite na profundidade consome tempo $O(Dn)$.

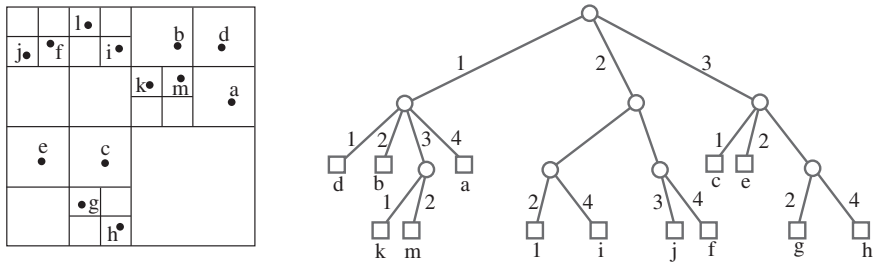


Figura 12.10 Uma *quadtree*. Demonstramos um exemplo de conjunto de pontos e a estrutura de *quadtree* correspondente.

Respondendo a consultas intervalares com uma *quadtree*

Uma das consultas normalmente respondidas pelas *quadtrees* é a consulta intervalar. Suponhamos que temos um retângulo A alinhado com o eixo das abscissas e iremos usar uma *quadtree* para retornar todos os pontos de S contidos em A. O método para responder a essas consultas é bastante simples. Começamos pela raiz r de T e comparamos a região R de r com A. Se A e R não se interseccionarem, então o algoritmo se encerra – não existem pontos na subárvore enraizada em r que caiam dentro de A. Por outro lado, se A contiver todo o R, então simplesmente enumeramos todos nodos externos descendentes de r. Esses são os dois casos simples. Se, por outro lado, R interseccionar A, mas A não contiver completamente R, então recursivamente executamos essa pesquisa em cada um dos filhos v de r.

Desempenho

Quando executamos essa consulta intervalar, no pior caso, podemos percorrer toda a árvore T e não produzir nenhuma saída. Assim, o tempo para executar uma consulta intervalar em uma *quadtree* de profundidade D, no pior caso, é $O(Dn)$. Do ponto de vista do pior caso, responder a uma consulta intervalar com uma *quadtree* é na verdade pior do que usando uma pesquisa de força bruta diretamente no conjunto S, o que con-

some um tempo $O(n)$ para responder a uma consulta bidimensional. Na prática, entretanto, a *quadtree* normalmente permite que as consultas sejam processadas mais rapidamente do que isso.

12.3.2 Árvores *k*-D

Existe um problema com *quadtrees*, que é o fato de que elas não são facilmente generalizáveis para dimensões maiores. Em particular, cada nodo em uma árvore de quatro dimensões análoga a uma *quadtree* poderá ter até 16 filhos. Cada nodo interno em uma *quadtree d*-dimensional pode ter até 2^d filhos. Para contornar o problema do grau muito alto no armazenamento de dados de dimensão maior do que três, os projetistas de estruturas de dados sempre consideram estruturas alternativas usando árvores de partição que sejam binárias.

Outro tipo de estrutura de dados de partição é a árvore *k*-D, que é similar à estrutura da *quadtree*, só que binária. A estrutura de dados da árvore *k*-D é, na verdade, uma família de árvores de partição, todas sendo árvores binárias de partição para armazenar dados multidimensionais. Da mesma forma que na estrutura de *quadtree*, cada nodo *v* de uma árvore *k*-D é associado com uma região retangular *R*, embora no caso da árvore *k*-D essa região não seja necessariamente quadrada. A diferença é que, na hora de executar uma operação de divisão de um nodo *v* em uma árvore *k*-D, isso é feito em relação a uma única linha perpendicular a um dos eixos coordenados. Para conjuntos tridimensionais ou de dimensões maiores, essa "linha" é um hiperplano alinhado a um dos eixos. Assim, não importando a dimensão, uma árvore *k*-D é sempre uma árvore binária, pois resolvemos a divisão associando a parte da região de *v* que está à "esquerda" da linha divisória com o filho da esquerda de *v* e a parte a "direita" com o filho *v* da direita. Igualmente a uma *quadtree*, suspendemos com as divisões quando o número de pontos que cai em uma região for menor do que um determinado limite (veja a Figura 12.11).

Existem dois tipos fundamentais de árvores *k*-D: árvores *k*-D ***baseadas em regiões*** e árvores *k*-D ***baseadas em pontos***. Árvores *k*-D baseadas em regiões são essencialmente versões binárias de *quadtrees*. Cada vez que uma região retangular *R* necessitar ser dividida em uma árvore *k*-D baseada em região, a região *R* é dividida exatamente ao meio por uma linha perpendicular ao lado mais longo de *R*. Se existir mais um lado mais longo, então eles são divididos seguindo a abordagem "*round-robin*". Por outro lado, árvores *k*-D baseadas em pontos executam as divisões baseadas na distribuição dos pontos dentro da região retangular. A árvore *k*-D da Figura 12.11 é baseada em pontos.

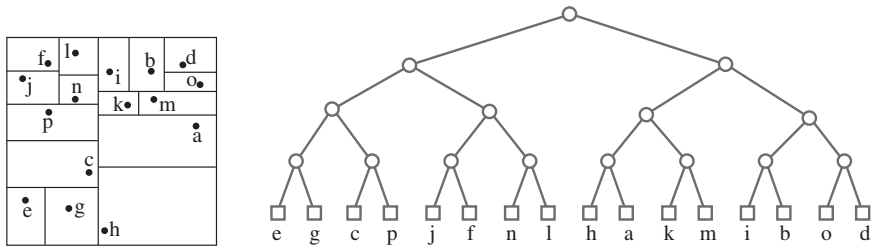


Figura 12.11 Um exemplo de árvore *k*-D.

O método para dividir um retângulo R que contem um subconjunto $S' \subseteq S$ de uma árvore k-D baseada em pontos envolve dois passos. No primeiro passo, determinamos a dimensão i que tiver a maior variação na dimensão i entre os pontos de S'. Isso pode ser feito, por exemplo, determinando, para cada dimensão j, os pontos de S' com valores mínimos e máximos para a dimensão j, e pegando i para ser a dimensão com maior intervalo entre esses dois valores. No segundo passo, determinamos o valor da dimensão mediana entre todos os pontos de S' e dividimos R com uma linha que atravesse esta mediana e seja perpendicular ao eixo i. Assim, a divisão de R divide o conjunto de pontos de S' na metade, mas pode não dividir a região R propriamente dita da mesma forma. Usando um método de tempo linear para determinar a mediana (Seção 4.7), essa etapa de divisão pode ser executada em tempo $O(k|S'|)$. Além disso, o tempo de execução de construção de uma árvore k-D para um conjunto de n pontos pode ser caracterizado pela seguinte equação recorrente: $T(n) = 2T(n/2) + kn$, que é $O(kn \log n)$. Além disso, uma vez que dividimos por dois o tamanho do conjunto de pontos associados com um nodo a cada divisão, a altura da árvore T é $\lceil \log n \rceil$. A Figura 12.11 ilustra uma árvore k-D baseada em pontos construída usando esse algoritmo.

A vantagem de uma árvore k-D baseada em pontos é a garantia de boa profundidade e bom tempo de construção. O problema do esquema baseado em pontos é que ele pode originar áreas retangulares "longas e magras", o que normalmente é considerado ruim para a maioria dos métodos de consulta sobre árvores k-D. Na prática, entretanto, tais regiões longas e magras são raras, e a maioria das regiões retangulares associadas com os nodos de uma árvore k-D tem forma de caixa.

Uso de árvores k-D para pesquisar o vizinho mais próximo

Vamos discutir como as árvores k-D podem ser usadas para responder a consultas. Em especial, vamos focar a pesquisa do vizinho mais próximo, na qual temos um ponto a ser consultado p e temos de encontrar o ponto de S que esteja mais próximo de p. Uma boa forma de usar uma árvore k-D, denotada por T, para responder a tal questão é como segue. Primeiramente pesquisamos a árvore T para localizar o nodo externo v com menor região retangular R que contém p. Todos os pontos de S que caírem em R ou na região associada com os irmãos de v são então comparados para encontrar o vizinho mais próximo, q. Podemos então definir uma esfera centrada em p que contenha q como sendo a esfera de menor vizinhança, s. Dada esta esfera, executamos então um caminhamento sobre T (usando, de preferência, um caminhamento de baixo para cima) para encontrar todas as regiões associadas com nodos externos de T que interseccionem s. Se, durante esse caminhamento encontrarmos um ponto mais próximo do que q, então atualizamos a referência q para que esta se refira a este novo ponto, e atualizamos a esfera s para conter o novo ponto. Não visitamos nenhum nodo cujas regiões não interseccionem s. Quando tivermos esgotado todas as alternativas, exibimos o ponto corrente q como o vizinho mais próximo de p. No pior caso, este método pode consumir tempo $O(n)$, mas existem várias análises experimentais e analíticas que sugerem que o tempo de execução médio aproxima-se de $O(\log n)$, usando algumas premissas razoáveis sobre a distribuição dos pontos de S. Além disso, existem algumas heurísticas úteis para acelerar essa pesquisa na prática, sendo que uma das melhores é a estratégia de pesquisa ***prioritária***, conforme a qual podemos explorar as subárvores de T na ordem da distância das regiões associadas a p.

12.4 A técnica do plano rotacional

Nesta seção, estudamos uma técnica que pode ser aplicada a diferentes problemas geométricos. A idéia principal é transformar um problema estático bidimensional em um problema dinâmico de uma dimensão, que pode ser resolvido usando uma seqüência de operações de inserção, remoção e consulta. Em vez de apresentar essa técnica em um conjunto abstrato, entretanto, demonstraremos seu uso com exemplos concretos.

12.4.1 Intersecção de segmentos ortogonais

O primeiro problema que resolveremos usando a técnica do plano rotacional é aquela que encontra todos os pares que se interseccionam em um conjunto de n segmentos de reta. É claro que podemos aplicar um algoritmo de força bruta para verificar todos os pares de segmentos para verificar se os mesmos se interseccionam. Uma vez que o número de pares é $n(n-1)/2$, este algoritmo consome tempo $O(n^2)$, pois podemos testar a intersecção de todos os pares em tempo constante. Se todos os pares se interseccionam, esse algoritmo é ótimo. Porém, podemos querer um método mais rápido para o caso em que o número de pares que se interseccionam for menor ou não houver intersecções. Se s for o número de pares que se interseccionam, gostaríamos de ter um algoritmo com saída sensível cujo tempo de execução fosse dependente de n e s. Apresentaremos um algoritmo que usa uma técnica de plano rotacional e executa em tempo $O(n \log n + s)$ para o caso em que o conjunto de segmentos de entrada consiste em ***segmentos ortogonais***, o que significa que cada segmento do conjunto é ou vertical ou horizontal.

Pesquisa intervalar unidimensional revisitada

Antes de continuarmos com nosso algoritmo, faremos uma pequena digressão para revisar um problema discutido anteriormente neste capítulo.

Referimo-nos problema da pesquisa intervalar unidimensional, no qual desejamos manter dinamicamente um dicionário de números (isto é, pontos sobre uma linha de números), sujeito a inserções, deleções e consultas, da seguinte forma:

findAllInRange(k_1,k_2): retorna a enumeração de todos os elementos de D com chave k, tal que $k_1 \leq k \leq k_2$.

Mostramos, na Seção 12.1.1, como podemos usar uma árvore de pesquisa binária balanceada, tal como uma árvore AVL ou uma árvore vermelho-preta, para manter tal dicionário de maneira a obter tempo $O(\log n)$ por inserção e remoção de ponto, e tempo $O(\log n + s)$ para responder a consultas findAllInRange, onde n é o número de pontos no dicionário no momento e s é o número de pontos retornados do intervalo. Não faremos uso dos detalhes deste algoritmo, apenas de sua existência; logo, o leitor que pulou a Seção 12.1.1 pode acreditar neste resultado e não precisa se preocupar sobre como foi alcançado.

Conjunto de problemas da pesquisa intervalar

Vamos retornar ao problema em questão, que é calcular todos os pares de segmentos que se interseccionam em uma coleção de segmentos horizontais e verticais. A idéia

principal do algoritmo de resolução deste problema é reduzir o problema bidimensional para um conjunto de problemas de pesquisa intervalar unidimensionais. Na verdade, para cada segmento vertical v, consideramos uma linha vertical $l(v)$ que passa por v, e mergulhamos no "mundo unidimensional" da linha $l(v)$ (veja as Figuras 12.13a e b). Apenas o segmento vertical v e as intersecções dos segmentos horizontais com $l(v)$ existem neste mundo. Em particular, o segmento v corresponde a um intervalo de $l(v)$; um segmento horizontal h interseccionando $l(v)$ corresponde a um ponto sobre $l(v)$ e os segmentos horizontais que cruzam v correspondem a pontos sobre $l(v)$ que estão contidos no intervalo.

Assim, se dispomos do conjunto $S(v)$ de segmentos horizontais que interseccionam $l(v)$, então determinar aqueles que interseccionam o segmento v é equivalente a executar uma pesquisa intervalar em $S(v)$ usando as ordenadas dos segmentos como chaves e o intervalo fornecido pelas ordenadas dos pontos finais do segmento v como intervalo de seleção.

Algoritmo de plano rotacional para intersecção de segmentos

Suponhamos que temos um conjunto de n segmentos verticais e horizontais no plano. Determinamos todos os pares de segmentos que se interseccionam neste conjunto usando a técnica de ***plano rotacional*** e a abordagem de conjuntos sugerida na idéia anterior. Este algoritmo envolve a simulação da rotação de uma linha vertical l sobre os segmentos movendo-se da direita para a esquerda, começando em uma posição à esquerda de todos segmentos informados. Durante a rotação, o conjunto de segmentos horizontais interseccionados pela linha rotacional é mantido em função das inserções e remoções de um dicionário ordenado pelas ordenadas. Quando a linha rotacional encontra um segmento vertical v, é executada uma consulta intervalar sobre o dicionário para localizar os segmentos horizontais que interseccionam v.

Mais especificamente, durante a rotação, mantemos um dicionário ordenado S armazenando os segmentos horizontais com suas chaves fornecidas pelas ordenadas. A rotação cessa quando da ocorrência de certos ***eventos*** que disparam as ***ações*** apresentadas na Tabela 12.12 e ilustradas na Figura 12.13.

Evento	**Ação**
Extremidade esquerda de um segmento horizontal h	Insere h no dicionário S
Extremidade direita de um segmento horizontal h	Remove h do dicionário S
Segmento vertical v	Executa uma pesquisa intervalar em S com o intervalo de seleção fornecido pelas ordenadas dos pontos extremos de v

Tabela 12.12 Eventos que disparam ações no algoritmo do plano rotacional para a intersecção de segmentos horizontais

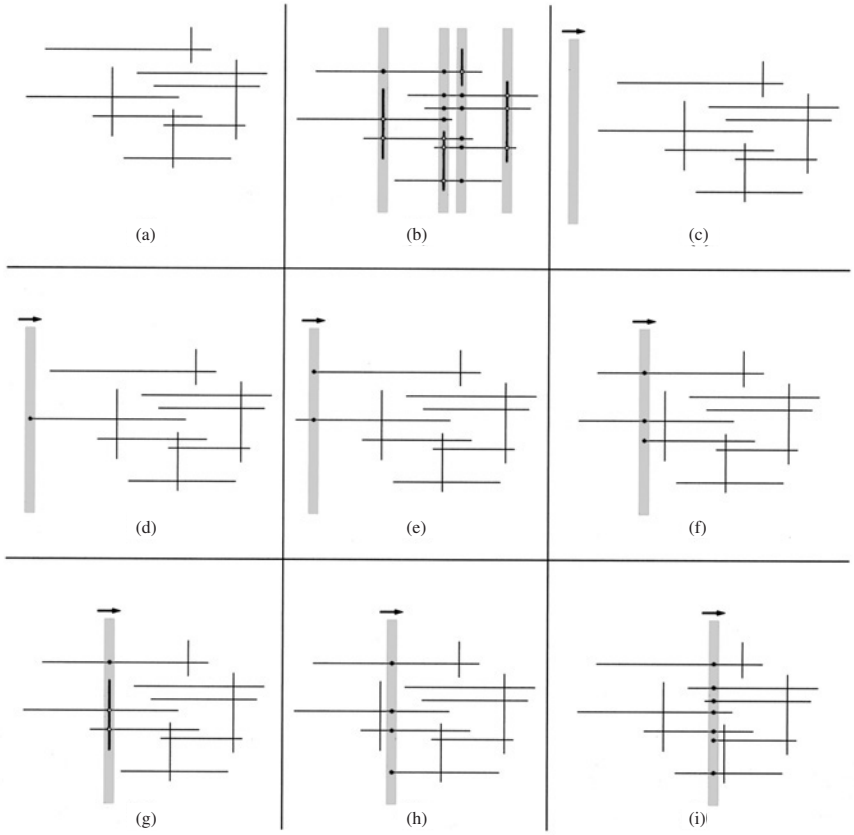


Figura 12.13 Plano rotacional para intersecção de segmentos: (a) conjunto de segmentos verticais e horizontais; (b) coleção de problemas de pesquisa intervalar unidimensionais; (c) início da rotação do plano (o dicionário ordenado S de segmentos horizontais está vazio); (d) o primeiro evento (extremidade esquerda) causa a intersecção em S; (e) o segundo evento (extremidade esquerda) causa outra inserção em S; (f) o terceiro evento (extremidade esquerda) causa mais uma inserção em S; (g) o primeiro segmento vertical causa uma pesquisa intervalar em S (duas intersecções relatadas); (h) o próximo evento (extremidade esquerda) causa uma inserção em S; (i) um evento na extremidade esquerda, três eventos mais tarde, causa uma inserção em S.

Desempenho

Para analisar o tempo de execução deste algoritmo de plano rotacional, observamos, inicialmente, que devemos identificar todos os eventos e ordená-los pelas abscissas. Um evento é um ponto extremo tanto de um segmento horizontal como de um segmento vertical. Assim, o número de eventos é no máximo $2n$. Quando ordenamos os eventos, comparamos os mesmos pela abscissa, o que consome tempo $O(1)$. Usando um dos algoritmos de ordenação assintoticamente ótimos, tais como o *heapsort* (Seção 2.4.4) ou o *merge-sort* (Seção 4.1), podemos ordenar os eventos em tempo $O(n \log n)$. As operações executadas no dicionário S são inserções, remoções e pesquisas intervalares. Cada vez que uma operação é executada, o tamanho de S é no máximo $2n$. Im-

plementamos S como uma árvore AVL (Seção 3.2) ou como uma árvore vermelho-preta (Seção 3.3.3), de maneira que as inserções e deleções consumam tempo $O(\log n)$. Como revisado anteriormente (e visto na Seção 12.1.1), a pesquisa intervalar em um dicionário ordenado de n elementos pode ser executada em $O(\log n + s)$ usando espaço $O(n)$, onde s é o número de itens relatados. Vamos caracterizar, então, o tempo de execução de uma pesquisa intervalar disparada por um segmento vertical v como sendo $O(\log n + s(v))$, onde $s(v)$ é o número de segmentos horizontais correntemente armazenados no dicionário S que interseccionam v. Assim, indicando o conjunto de segmentos verticais com V, o tempo de execução da rotação é

$$O\left(2n\log n + \sum_{v\in V}(\log n + s(v))\right)$$

Uma vez que a rotação atravessa todos os segmentos, a soma de $s(v)$ para todos os segmentos encontrados é igual ao número total s de pares de segmentos que se interseccionam. Assim, concluímos que a rotação consome tempo $O(n \log n + s)$.

Resumindo, o algoritmo completo de intersecção de segmentos descrito consiste em uma etapa de ordenação de eventos, seguido por uma etapa de rotação. Ordenar os eventos consome tempo $O(n \log n)$, enquanto que a rotação consome tempo $O(n \log n + s)$. Logo, o tempo de execução do algoritmo é $O(n \log n + s)$.

12.4.2 Determinação do par de pontos mais próximo

Outro problema geométrico que pode ser resolvido usando a técnica do plano rotacional envolve o conceito de ***proximidade***, que é a relação da ***distância*** que existe entre objetos geométricos. Em particular, focamos o problema do ***par mais próximo***, que consistem em encontrar um par de pontos p e q que estejam a uma distância mínima um do outro em um conjunto de n pontos. Esse par é conhecido como o mais próximo. Iremos usar a definição euclidiana de distância entre dois pontos a e b:

$$\mathsf{dist}(a,b) = \sqrt{(x(a)-x(b))^2 + (y(a)-y(b))^2},$$

onde $x(p)$ e $y(p)$ denotam, respectivamente, as abscissas e as ordenadas do ponto p. Aplicações do problema do par mais próximo incluem a verificação de partes mecânicas e circuitos integrados, nos quais é importante que certas regras de separação entre componentes sejam respeitadas.

Um algoritmo de plano rotacional

Um algoritmo direto de "força bruta" para resolver o problema do par mais próximo consiste em calcular a distância entre cada par de pontos e selecionar o par com a distância mínima. Uma vez que o número de pares é $n(n-1)/2$, esse algoritmo consome tempo $O(n^2)$. Podemos, entretanto, aplicar uma estratégia mais inteligente que evite a verificação de todos os pares de pontos.

Verificamos que podemos aplicar efetivamente a técnica do plano rotacional ao problema do par mais próximo. Resolvemos esse problema, neste caso, imaginando que rotacionamos o plano por uma linha vertical da esquerda para direita, iniciando na posição à esquerda de todos os n pontos de entrada. À medida que rotamos a linha através do plano, seguimos o par mais próximo encontrado até então e todos os pontos que estão "próximos" da linha de rotação. Também mantemos a distância d entre o par mais próximo encontrado até então. Em especial, como demonstramos na Figura 12.14, enquanto rotamos pelos pontos da esquerda para a direita, mantemos os seguintes dados:

- o par mais próximo (a,b) entre os pontos encontrados e a distância $d = dist(a,b)$;
- um dicionário ordenado S que armazena os pontos que caem em uma faixa de largura d à esquerda da linha de rotação e usa as ordenadas dos pontos como chaves.

Cada ponto entrado p corresponde a um evento neste plano rotacional. Quando a linha de rotação encontra um plano p, executamos as seguintes ações:

1. Atualizamos o dicionário S removendo os pontos com distância horizontal maior do que d de p, isto é, todo ponto r onde $x(p) - x(r) > d$.
2. Encontramos o ponto mais próximo q à esquerda de p pesquisando no dicionário S (diremos em seguida como isso é feito). Se $\mathsf{dist}(p,q) < d$, então atualizamos o mais próximo par corrente e a distância fazendo $a \leftarrow p$, $b \leftarrow q$ e $d \leftarrow \mathsf{dist}(p,q)$.
3. Inserimos p em S.

Claramente podemos restringir nossa pesquisa do ponto mais próximo q à esquerda de p aos pontos no dicionário S, uma vez que todos os outros pontos terão distância maior do que d. O que queremos são aqueles pontos de S que caem no maior círculo $C(p,d)$ de raio d centrado e à esquerda do ponto p (veja a Figura 12.15). Como primeira aproximação, pegamos os pontos $B(p,d)$ de $C(p,d)$ que estão dentro de uma caixa retangular de $d \times 2d$ (Figura 12.15) executando uma pesquisa intervalar (Seção 12.11) sobre S para os pontos de S com ordenadas no intervalo de chaves $[y(p) - d, y(p) + d]$. Examinamos tais pontos, um a um, e encontramos o que está mais próximo de p, denotado por q. Como as operações executadas sobre o dicionário S são pesquisas intervalares, inserções e remoções de pontos, implementamos S em função de uma árvore AVL ou uma árvore vermelho-preta.

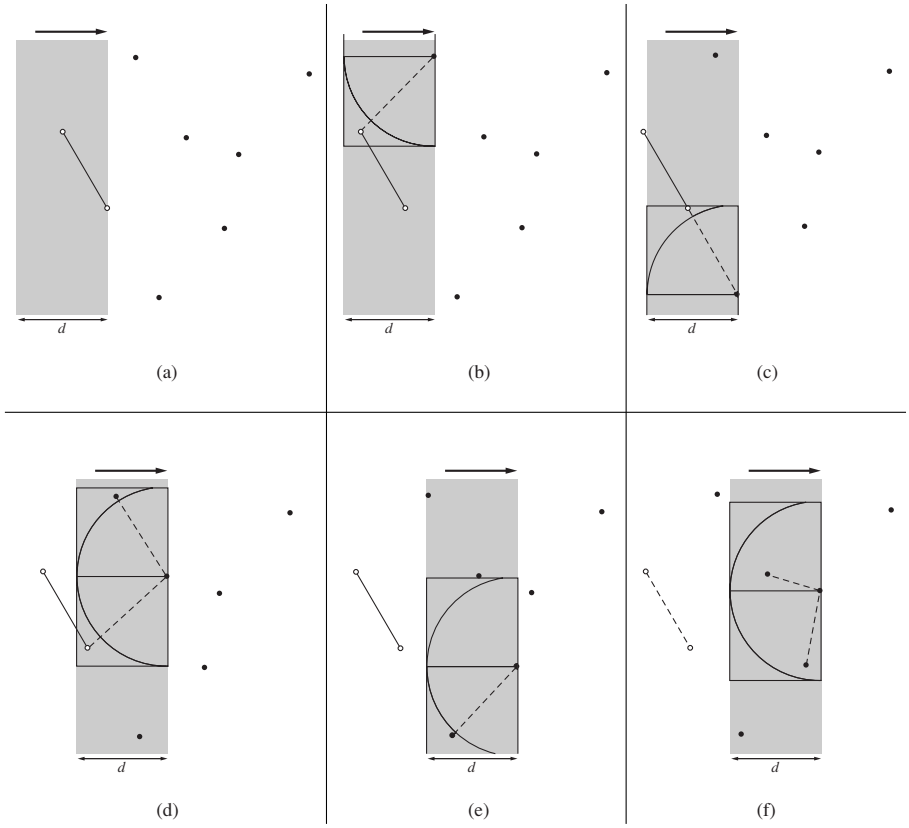


Figura 12.14 Plano rotacional para o problema do par mais próximo: (a) a distância mínima inicial d e o par mais próximo (destacado); (b) o próximo evento (a caixa $B(p, d)$ contém um ponto, mas o semicírculo $C(p, d)$ está vazio); (c) próximo evento ($C(p, d)$ está novamente vazio, mas um ponto é removido de S); (d) próximo evento (com $C(p, d)$ ainda vazio). O dicionário S contém os pontos na faixa cinza com largura d; (e) um ponto p é encontrado (e um ponto é removido de S), com $B(p, d)$ contendo um ponto, mas $C(p, d)$ não contendo nenhum; logo, a distância mínima d e o par mais próximo (a, b) permanecem os mesmos; (f) um ponto p é encontrado e $C(p, d)$ contém dois pontos (e um ponto é removido de S).

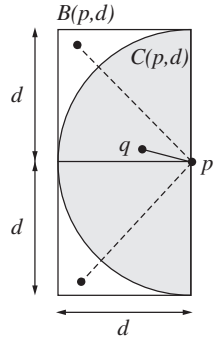


Figura 12.15 A caixa $B(p, d)$ e o semicírculo $C(p, d)$.

A seguinte propriedade intuitiva, cuja prova fica como exercício (R-12.3), é crucial para a análise do tempo de execução do algoritmo.

Teorema 12.7: *um retângulo de largura d e altura 2d pode conter no máximo seis pontos de maneira que quaisquer dois pontos estejam a uma distância mínima d.*

Logo, existem no máximo seis pontos de S que caem na caixa $B(p, d)$. Assim, a pesquisa intervalar em S para encontrar os pontos em $B(p, d)$ consome tempo $O(\log n + 6)$, que corresponde a $O(\log n)$. Além disso, podemos encontrar o ponto em $B(p, d)$ mais próximo de p em tempo $O(1)$.

Antes de começarmos a rotação, ordenamos os pontos por suas abscissas e os armazenamos em uma lista ordenada X. A lista X é usada para dois propósitos:

- obter o ponto seguinte a ser processado;
- identificar os pontos a serem removidos do dicionário S.

Fazemos referência a duas posições na lista X, que denotamos por firstInStrip e lastInStrip. A posição lastInStrip mantém o novo ponto a ser inserido em S, enquanto que a posição firstInStrip mantém o ponto mais à esquerda de S. Avançando em lastInStrip uma etapa por vez, encontramos um novo ponto a ser processado. Usando firstInStrip, identificamos os pontos a serem removidos de S. Na verdade, enquanto temos

$$x(\text{point(firstInStrip)}) < x(\text{point(lastInStrip)} - d,$$

executamos a operação removeElement(y(point(firstInStrip))) sobre o dicionário S e avançamos firstInStrip.

Seja n o número de pontos de entrada. Nossa análise do algoritmo de plano rotacional para o problema do par mais próximo é baseado nas seguintes observações:

- A ordenação preliminar pela abscissas consome tempo $O(n \log n)$.
- Cada ponto é inserido uma vez e removido uma vez do dicionário S que tem tamanho máximo n; assim, o tempo total para inserir e remover elementos de S é $O(n \log n)$.
- Pelo Teorema 12.7, cada consulta intervalar em S consome tempo $O(\log n)$. Executamos essa consulta intervalar toda vez que processamos um ponto novo. Assim, o tempo total gasto para executar consultas intervalares é $O(n \log n)$.

Concluímos, assim, que podemos calcular o par mais próximo de um conjunto de n pontos em tempo $O(n \log n)$.

12.5 Cobertura convexa

Um dos problemas geométricos mais estudados é o cálculo da cobertura convexa de um conjunto de pontos. De uma maneira informal, a ***cobertura convexa*** de um conjunto de pontos no plano é a forma obtida por uma faixa elástica colocada "ao redor dos pontos", permitindo que a mesma se encolha até atingir um estado de equilíbrio (veja a Figura 12.16).

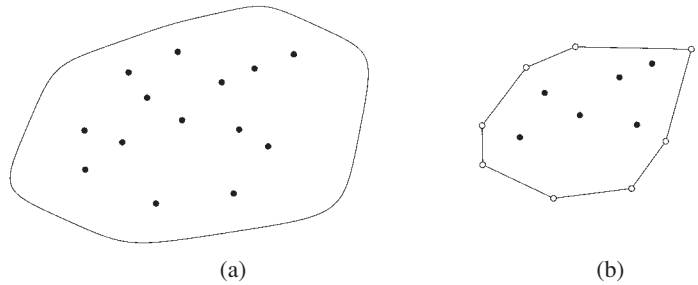


Figura 12.16 Cobertura convexa de um conjunto de pontos no plano: (a) um exemplo de faixa elástica colocada ao redor dos pontos; (b) a cobertura convexa dos pontos.

A cobertura convexa corresponde, intuitivamente, à noção de limite de um conjunto de pontos, e pode ser usada para aproximar a forma de um objeto complexo. Na verdade, o cálculo da cobertura convexa de um conjunto de pontos é uma operação fundamental em geometria computacional. Antes de detalhar, porém, a cobertura convexa e os algoritmos para calculá-la, precisamos primeiramente discutir algumas premissas de representação para objetos geométricos.

12.5.1 Representação de objetos geométricos

Algoritmos geométricos lidam com objetos geométricos de vários tipos como entrada. A geometria básica de objetos planos são pontos, linhas, segmentos e polígonos.

Existem muitas formas de representar objetos geométricos planos. Em vez de fornecer TADs diferentes para pontos, linhas, segmentos e polígonos, como seria apropriado para um livro de algoritmos geométricos, assumiremos que temos representações intuitivas para esses objetos. Sendo assim, mencionamos brevemente algumas das escolhas que podemos fazer em relação a representações geométricas.

Podemos representar um ponto no plano por um par (x,y) que armazena as coordenadas cartesianas x e y do ponto. Embora essa representação seja versátil, não é a única. Existem algumas aplicações em que representações diferentes podem ser melhores (tais como representar um ponto como a intersecção entre duas linhas não-paralelas).

Linhas, segmentos e polígonos

Podemos representar uma linha l como uma tripla (a, b, c) em que esses valores correspondam aos coeficientes a, b e c da equação linear

$$ax + by + c = 0$$

associada com l. Por outro lado, em vez disso, podemos especificar dois pontos diferentes q_1 e q_2, e associá-los com a linha que passa por ambos. Dadas as coordenadas cartesianas (x_1, y_1) de q_1 e (x_2, y_2) de q_2, a equação da linha l que passa por q_1 e q_2 é dada por

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} ,$$

de onde derivamos

$$a = (y_2 - y_1);\ b = -(x_2 - x_1);\ c = y_1\,(x_2 - x_1) - x_1\,(y_2 - y_1).$$

Um segmento de linha s é tipicamente representado por um par (p,q) de pontos no plano que formam as extremidades de s. Podemos também representar s fornecendo a linha que passa por ele, juntamente a um intervalo de abscissas e ordenadas que restrinjam essa linha ao segmento s. (Por que não é suficiente incluir apenas o intervalo da abscissas ou das ordenadas?)

Podemos representar um polígono P por uma seqüência circular de pontos, chamados de ***vértices*** de P (veja a Figura 12.7). Os segmentos entre os vértices consecutivos de P são chamados de ***arestas*** de P. O polígno P é chamado de ***não-interseccionado*** ou ***simples***, se intersecções entre pares de arestas de P ocorrerem apenas em vértices correspondentes a extremidades comuns. Um polígono é ***convexo*** se for simples e todos os seus ângulos internos forem menores do que π.

A discussão sobre as diferentes maneiras de representar pontos, linhas, segmentos e polígonos não pretende ser exaustiva, mas tenciona apenas indicar diferentes maneiras de implementar esses objetos geométricos.

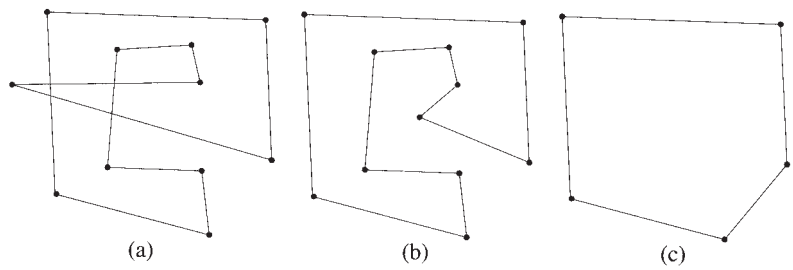


Figura 12.17 Exemplos de polígonos: (a) interseccionado, (b) simples e (c) convexo.

12.5.2 Teste da orientação dos pontos

Uma relação geométrica importante, explorada em muitos algoritmos geométricos, e em especial na construção de uma cobertura convexa, é a ***orientação***. Dada uma tripla ordenada (p, q, r) de pontos, dizemos que (p, q, r) ***vira à esquerda*** e é orientada no sentido ***anti-horário*** se o ângulo formado no lado esquerdo quando se vai de p para q e então para r for menor do que π. Se o ângulo do lado esquerdo for menor do que π, entretanto, então dizemos que (p, q, r) **vira à direita** e é orientada no sentido ***horário*** (veja a Figura 12.18). É possível que os ângulos do lado esquerdo e do direito sejam ambos iguais a π, no caso em que os mesmos não viram para lado nenhum, sendo sua orientação ***colinear***.

Dada uma tripla (p_1, p_2, p_3) de três pontos $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ e $p_3 = (x_3, y_3)$ no plano, faça $\Delta(p_1, p_2, p_3)$ ser o determinante definido por

$$\Delta(p_1, p_2, p_3) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 - x_2y_1 + x_3y_1 - x_1y_3 + x_2y_3 - x_3y_2. \qquad (12.1)$$

A função $\Delta(p_1, p_2, p_3)$ é normalmente chamada de função da "área sinalizada", porque seu valor absoluto é o dobro da área do (possivelmente degenerado) triângulo formado pelos pontos p_1, p_2 e p_3. Além disso, temos o seguinte fato importante relacionando essa função ao teste de orientação.

Teorema 12.8: *a orientação de uma tripla* (p_1,p_2,p_3) *de pontos no plano pode ser anti-horária, horária ou colinear, dependendo de* $\Delta(p_1,p_2,p_3)$ *ser positivo, negativo ou zero, respectivamente.*

Esboçaremos a prova do Teorema 12.18; deixamos os detalhes como exercício (R-12.4). Na Figura 12.18, mostramos uma tripla (p_1, p_2, p_3) de pontos tais que $x_1 < x_2 < x_3$. Claramente, essa tripla vira à esquerda se a inclinação do segmento p_2p_3 for maior do que a inclinação do segmento p_1p_2. Isso é expresso pela seguinte questão:

$$\text{É } \frac{y_3 - y_2}{x_3 - x_2} > \frac{y_2 - y_1}{x_2 - x_1} \ ? \tag{12.2}$$

Pela expansão de $\Delta(p_1, p_2, p_3)$ mostrada em 12.1, podemos verificar que a desigualdade 12.2 é equivalente a $\Delta(p_1, p_2, p_3) > 0$.

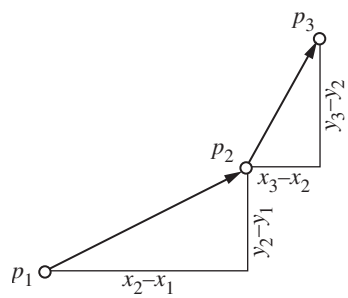


Figura 12.18 Um exemplo de uma virada à esquerda. As diferenças entre as coordenada de p_1 e p_2 e entre p_2 e p_3 também são ilustradas.

Exemplo 12.9: *usando a noção de orientação, vamos considerar o problema de testar quando dois segmentos de linha* s_1 *e* s_2 *se interseccionam. Mais precisamente, faça* $s_1 = \overline{p_1q_1}$ *e* s_2 $\overline{p_2q_2}$= *serem dois segmentos no plano.* s_1 *e* s_2 *se interseccionam se e somente se* ***uma das*** *seguintes condições se verificar:*

1. (a) (p_1,q_1,p_2) *e* (p_1,q_1,q_2) *têm diferentes orientações* ***e***
   (b) (p_2,q_2,p_1) *e* (p_2,q_2,q_1) *têm diferentes orientações.*
2. (a) (p_1,q_1,p_2), (p_1,q_1,q_2), (p_2,q_2,p_1) *e* (p_2,q_2,q_1) *são todos colineares,* ***e***
   (b) *as projeções das abscissas de* s_1 *e* s_2 *se interseccionam* ***e***
   (c) *as projeções das ordenadas de* s_1 *e* s_2 *se interseccionam.*

A Condição 1 *é ilustrada na Figura* 12.19. *Também mostramos, na Tabela* 12.20, *a orientação respectiva das triplas* (p_1, q_1, p_2), (p_1, q_1, q_2), (p_2, q_2, p_1) *e* (p_2, q_2, q_1) *em cada um dos quatro casos para a Condição* 1. *Uma prova completa é deixada como exercício* (*R*-12.5). *Observamos que as condições também se aplicam se* s_1 *e/ou* s_2 *for um segmento degenerado com extremidades coincidentes.*

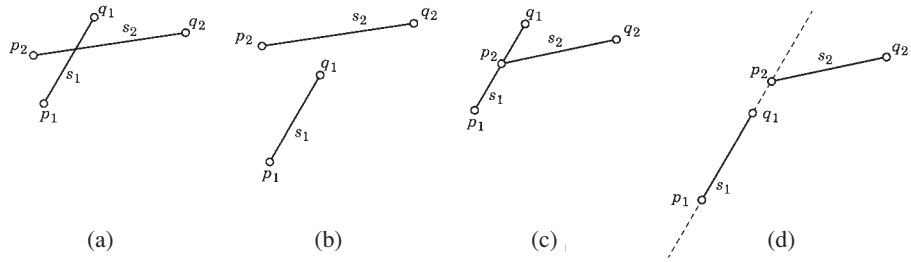


Figura 12.19 Exemplos demonstrando os quatro casos da Condição 1 do Exemplo 12.9.

Caso	(p_1, q_1, p_2)	(p_1, q_1, q_2)	(p_2, q_2, p_1)	(p_2, q_2, q_1)	Intersecção?
(a)	SAH	SH	SH	SAH	sim
(b)	SAH	SH	SH	SH	não
(c)	COL	SH	SH	SAH	sim
(d)	COL	SH	SH	SH	não

Tabela 12.20 Os quatro casos apresentados na Figura 12.19 para as orientações especificadas pela Condição 1 do exemplo 12.9, em que SAH indica sentido anti-horário, SH indica sentido horário e COL indica colinearidade.

12.5.3 Propriedades básicas de coberturas convexas

Dizemos que a região R é ***convexa*** se quaisquer dois pontos p e q estiverem em R; então, o segmento de linha $\overline{pq}$ está todo contido em R. A ***cobertura convexa*** de um conjunto de pontos S é o limite da menor região convexa que contém todos os pontos de S em seu interior ou sobre seus limites. A noção de "menor" refere-se tanto ao perímetro como à área da região, sendo que ambas as definições são equivalentes. A cobertura convexa de um conjunto de pontos S no plano define um polígono convexo, e os pontos de S no limite da cobertura convexa definem os vértices deste polígono. O exemplo a seguir descreve uma aplicação do problema da cobertura convexa em um planejamento de movimentação para robôs.

Exemplo 12.10: *um problema comum em robótica é identificar uma trajetória partindo de um ponto s até um destino t evitando um certo obstáculo. Entre as diferentes trajetórias possíveis, queremos descobrir a menor possível. Vamos assumir que o obstáculo é um polígono P. Podemos calcular a menor trajetória de s até t, evitando P com a seguinte estratégia* (*veja a Figura* 12.21).

- *Determinamos se o segmento de reta $\ell = \overline{st}$ intersecciona P. Se não intersecciona, então ℓ é a menor trajetória que evita P.*
- *Em qualquer outro caso, se $\overline{st}$ interseccionar P, então calculamos a cobertura convexa H dos vértices do polígono P mais os pontos s e t. Observamos que s e t subdividem a cobertura convexa H em duas seqüências poligonais, uma avançando no sentido horário de s até t e outra no sentido anti-horário de s até t.*
- *Selecionamos e retornamos a menor das duas seqüências com pontos extremos s e t que se localize sobre H.*

A menor seqüência é o menor caminho sobre o plano que evita o obstáculo P.

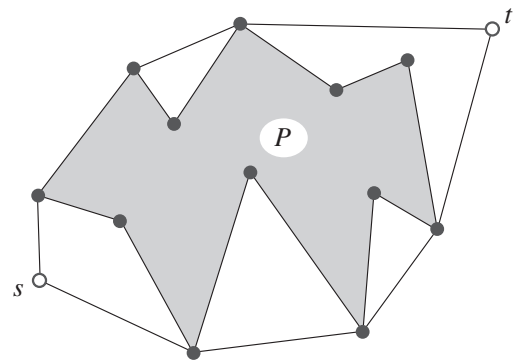


Figura 12.21 Um exemplo da menor trajetória de um ponto s até p que evita um obstáculo poligonal P; a menor trajetória é a seqüência horária de s até t.

Existem várias aplicações para o problema da cobertura convexa, incluindo problemas de particionamento, de teste de formas e de separação. Por exemplo, para determinar se existe um semiplano (isto é, a região do plano de um lado de uma reta) que contém completamente um conjunto de pontos A, mas evita totalmente um conjunto de pontos B, é suficiente calcular as coberturas convexas de A e B e determinar se as mesmas se interseccionam.

Existem várias propriedades geométricas interessantes associadas com coberturas convexas. O teorema a seguir provê uma caracterização alternativa dos pontos que pertencem a uma cobertura convexa e dos que não pertencem.

Teorema 12.11: *faça S ser um conjunto de pontos planares com cobertura convexa H. Então:*

- *Um par de pontos a e b de S formam uma aresta de H se e somente se todos os pontos de S estiverem contidos em um lado da linha que passa por a e b.*
- *Um ponto p de S é um vértice de H se e somente se existir uma linha l que passe por p, de maneira que todos os outros pontos de S estejam contidos no mesmo semiplano delimitado por l (isto é, eles estão todos do mesmo lado de l).*
- *Um ponto p de S não é vértice de H se e somente se p estiver contido no interior de um triângulo formado por três outros pontos de S ou no interior de um segmento formado por dois pontos de S.*

As propriedades expressas pelo Teorema 12.11 são demonstradas na Figura 12.22. Uma prova completa das mesmas é deixada como exercício (R-12.6). Como conseqüência do Teorema 12.11, podemos verificar imediatamente que, em qualquer conjunto S de pontos no plano, os seguintes pontos ***críticos*** estão sempre no limite da cobertura convexa de S:

- um ponto com abscissa mínima,
- um ponto com abscissa máxima,
- um ponto com ordenada mínima,
- um ponto com ordenada máxima.

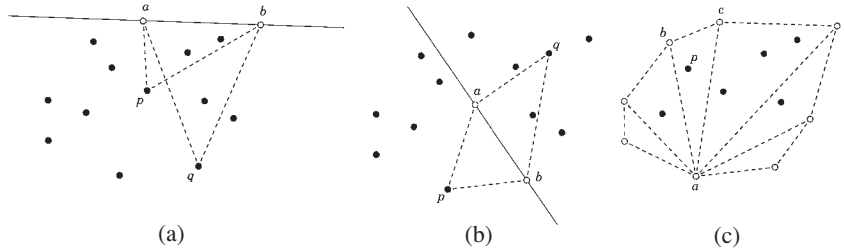


Figura 12.22 Demonstração das propriedades da cobertura convexa apresentadas no Teorema 12.11: (a) pontos *a* e *b* compõem uma aresta da cobertura convexa; (b) pontos *a* e *b* não formam uma aresta da cobertura convexa; (c) o ponto *p* não está sobre a cobertura convexa.

12.5.4 O algoritmo do embrulho para presente

O Teorema 12.11 declara, basicamente, que podemos identificar um ponto específico, digamos um com ordenada mínima, que nos forneça uma configuração inicial para o algoritmo que calcula a cobertura convexa. O algoritmo do ***embrulho para presente*** para calcular a cobertura convexa de um conjunto de pontos no plano é baseado apenas em tal ponto inicial e pode ser descrito intuitivamente como segue (veja a Figura 12.23):

1. Imaginemos os pontos como pinos fixados em um determinado nível e que amarramos um cabo no pino que corresponde ao ponto *a* com ordenada mínima (e abscissa mínima, se existirem laços). Chamamos *a* de ***ponto de ancoragem*** e observamos que *a* é um vértice da cobertura convexa.
2. Puxemos o cabo para a direita do ponto de ancoragem e rotamos o mesmo no sentido anti-horário até que este toque em outro pino que corresponde ao próximo vértice da cobertura convexa.
3. Continuemos rotando o cabo no sentido anti-horário, identificando um novo vértice da cobertura a cada passo, até que o cabo volte a encontrar o pino inicial.

Cada vez que rotamos nosso "cabo" ao redor do pino atual até este encostar em outro ponto, executamos uma operação chamada de etapa de ***empacotamento*** (ou embrulho). Geometricamente, uma etapa de empacotamento envolve iniciar em uma linha *L* conhecida por ser tangente à cobertura convexa no ponto atual de ancoragem *a*, e determinar a linha que passa por *a* e por outro ponto do conjunto que faça o menor ângulo com *L*. Implementar essa etapa de empacotamento não requer, entretanto, funções trigonométricas e cálculos de ângulos. Em vez disso, podemos executar uma etapa de empacotamento nos termos do seguinte teorema, que resulta do Teorema 12.11.

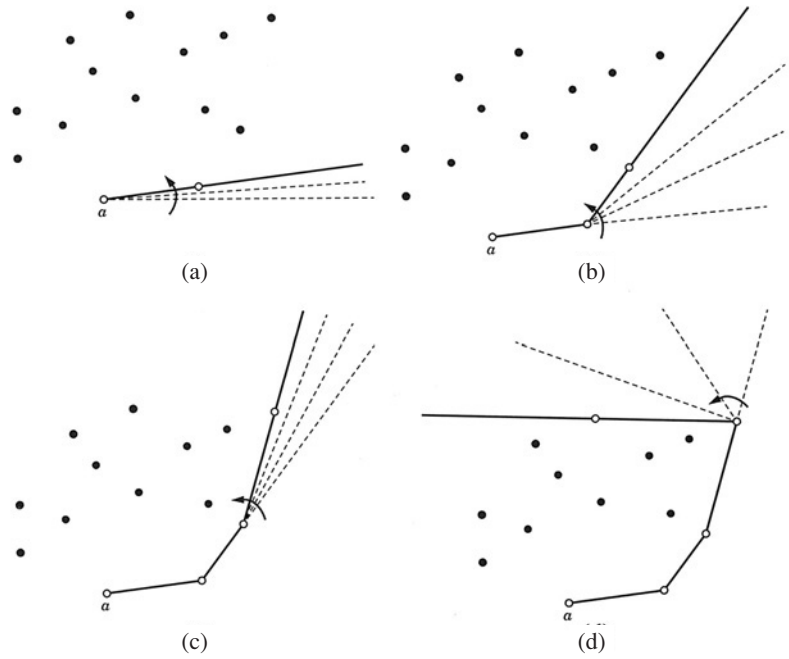


Figura 12.23 Quatro etapas iniciais do empacotamento do algoritmo do embrulho para presente.

Teorema 12.12: *faça S ser um conjunto de pontos no plano, e faça a ser um ponto de S que seja um vértice da cobertura convexa H de S. O próximo vértice de H, andando no sentido anti-horário a partir de a, é o ponto p, de forma que a tripla* (*a*,*p*,*q*) *vire à esquerda a cada ponto q de s.*

Relembrando a discussão da Seção 2.4.1, vamos definir um comparador $C(a)$ que use a orientação de (a,p,q) para comparar dois pontos p e q de S. Isto é, $C(a)$.isLess(p,q) retorna verdadeiro se a tripla (a,p,q) virar à esquerda. Chamamos o comparador $C(a)$ de comparador ***radial***, na medida em que compara pontos em termos de suas relações radiais ao redor do ponto de ancoragem a. Pelo Teorema 12.12, o vértice que segue a no sentido anti-horário na cobertura é simplesmente o ponto mínimo em relação ao comparador radial.

Desempenho

Podemos agora analisar o tempo de execução do algoritmo do embrulho para presente. Seja n o número de pontos de S e seja $h \leq n$ o número de vértices da cobertura convexa H de S. Faça $p_0,..., p_{h-1}$ ser o número de vértices de H. Encontrar o ponto de ancoragem $a = p_0$ consome tempo $O(n)$. Uma vez que em cada etapa de empacotamento do algoritmo descobrimos um novo vértice da cobertura convexa, o número de etapas de empacotamento é igual a h. A etapa i é um cálculo de procura mínima baseado no comparador radial $C(p_{i-1})$, que executa em tempo $O(n)$, uma vez que a determinação da orientação da tripla consome tempo $O(1)$ e que devemos examinar todos os pontos

de S para encontrar o menor em relação a $C(p_{i-1})$. Concluímos que o algoritmo do embrulho para presente executa em tempo $O(hn)$, que é $O(n^2)$ no pior caso. Na verdade, o pior caso do algoritmo do embrulho para presente ocorre quando $h = n$, isto é, quando todos os pontos estão na cobertura convexa.

O tempo para o pior caso do algoritmo do embrulho para presente em função de n, por isso, não é muito eficiente. Esse algoritmo é, entretanto, razoavelmente eficiente na prática, porque tira vantagem da situação (mais comum) quando h, o número de pontos da cobertura, for menor em relação ao número de pontos de entrada, n. Isto é, este algoritmo é um algoritmo ***sensível à saída*** – um algoritmo cujo tempo de execução depende do tamanho da saída. O embrulho para presente tem um tempo de execução que varia entre linear e quadrático e é eficiente quando a cobertura convexa tem poucos vértices. Na próxima seção, veremos um algoritmo que é eficiente para todos os tamanhos de cobertura, apesar de ser um pouco mais complicado.

12.5.5 O algoritmo de varredura de Graham

Um algoritmo de cobertura convexa que tem um tempo de execução eficiente, não importando quantos pontos estão dentro do limite da cobertura, é o algoritmo de ***varredura de Graham***. O algoritmo de varredura de Graham para calcular a cobertura convexa H de um conjunto P de n pontos no plano consiste nas três fases a seguir:

1. Determinamos um ponto a de P que seja vértice de H e o chamamos de ***ponto de ancoragem***. Podemos, por exemplo, escolher como ponto de ancoragem a o ponto de P com menor ordenada (e menor abscissa, se existirem laços).
2. Ordenamos os pontos restantes de P (isto é, $P - \{a\}$) usando o comparador radial $C(a)$, e fazemos S ser a lista ordenada de pontos resultante (veja a Figura 12.24). Na lista S, os pontos de P aparecem ordenados em sentido anti-horário "por ângulo" em relação ao ponto de ancoragem a, ainda que nenhuma computação de ângulos seja executada pelo comparador.

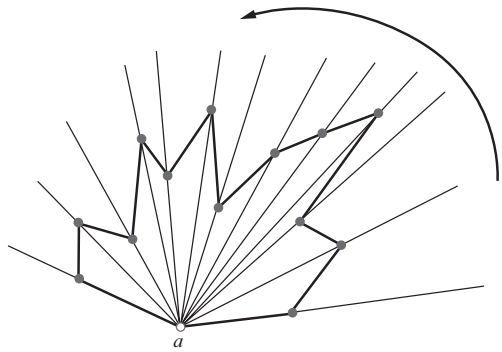


Figura 12.24 Ordenação ao redor do ponto de ancoragem no algoritmo de varredura de Graham.

3. Após acrescentar o ponto de ancoragem a na primeira e na última posição de S, ***varremos*** os pontos de S em ordem (radial), mantendo, a cada etapa, uma lista H armazenando uma cadeia convexa que envolve os pontos percorridos. Cada vez que consideramos um novo ponto p, executamos o seguinte teste:

   (a) Se p formar uma curva à esquerda com os últimos dois pontos de H, ou se H contiver menos de dois pontos, então adicione p ao fim de H.

   (b) Em qualquer outro caso, remova o último ponto de H e repita o teste para p.

   Paramos quando retornamos o ponto de ancoragem a, no qual o ponto H armazena os vértices da cobertura convexa de P em sentido horário.

Os detalhes da fase de varredura (Fase 3) são detalhados no algoritmo Scan, descrito no Algoritmo 12.25 (veja a Figura 12.26).

Algoritmo Scan(S,a):
Entrada: uma lista S de pontos no plano iniciando com o ponto a, sendo que a está dentro da cobertura convexa de S e os pontos restantes estão ordenados em sentido anti-horário ao redor de a.
Saída: lista S contendo apenas os vértices da cobertura convexa.

```
S.insertLast(a)          {adiciona uma cópia de a no final de S}
prev ← S.first()         {então, inicialmente prev = a}
curr ← S.after(prev)             {o próximo ponto está na cadeia convexa atual}
repita
  next ← S.after(curr)               {avança}
  se ponto(point(prev), point(curr), point(next)) fazem uma curva para a esquerda
  então
    prev ← curr
  senão
    S.remove(curr)               {o ponto curr não está na cobertura convexa}
    prev ← S.before(prev)
  curr ← S.after(prev)
até curr = S.last()
S.remove(S.last())             {remove a cópia de a}
```

Algoritmo 12.25 Fase de varredura do algoritmo de Graham para determinar a cobertura convexa (veja a Figura 12.26). As variáveis *prev*, *curr* e *next* são posições (Seção 2.2.2) da lista S. Assumimos que o método de acesso point(*pos*) está definido e retorna o ponto armazenado na posição *pos*. Fornecemos uma descrição simplificada do algoritmo que trabalha apenas se S tiver pelo menos três pontos e não existirem três pontos colineares.

Desempenho

Vamos agora analisar o tempo de execução do algoritmo de varredura de Graham. Denotamos o número de pontos em P (e S) com n. A primeira fase (encontrar o ponto de ancoragem) claramente consome tempo $O(n)$. A segunda fase (ordenar os pontos ao redor do ponto ancora) consome tempo $O(n\log n)$, desde que usemos um algoritmo de ordenação assintoticamente ótimo, tal como o *heapsort* (Seção 2.4.4) ou o *merge-sort* (Seção 4.1). A análise da (terceira) fase de varredura é mais delicada.

Para analisar a fase de varredura do algoritmo de Graham, vamos examinar com mais detalhes o laço ***repita*** do Algoritmo 12.25. A cada iteração do laço, a variável *next* pode avançar uma posição na lista S (se o teste ***se*** for bem-sucedido) ou a variável *next* permanece na mesma posição, mas, neste caso, um ponto é removido de S (se o teste ***se*** não for bem-sucedido). Assim, o número de iterações do laço ***repita*** é no máximo $2n$. Assim, cada sentença do algoritmo Scan é executada no máximo $2n$ vezes. Uma vez que cada sentença requer a execução de $O(1)$ operações elementares, o algoritmo Scan consome tempo $O(n)$. Concluindo, o tempo de execução do algoritmo de varredura de Graham é dominado pela segunda fase, em que a ordenação é executada. Assim, o algoritmo de varredura de Graham executa em tempo $O(n \log n)$.

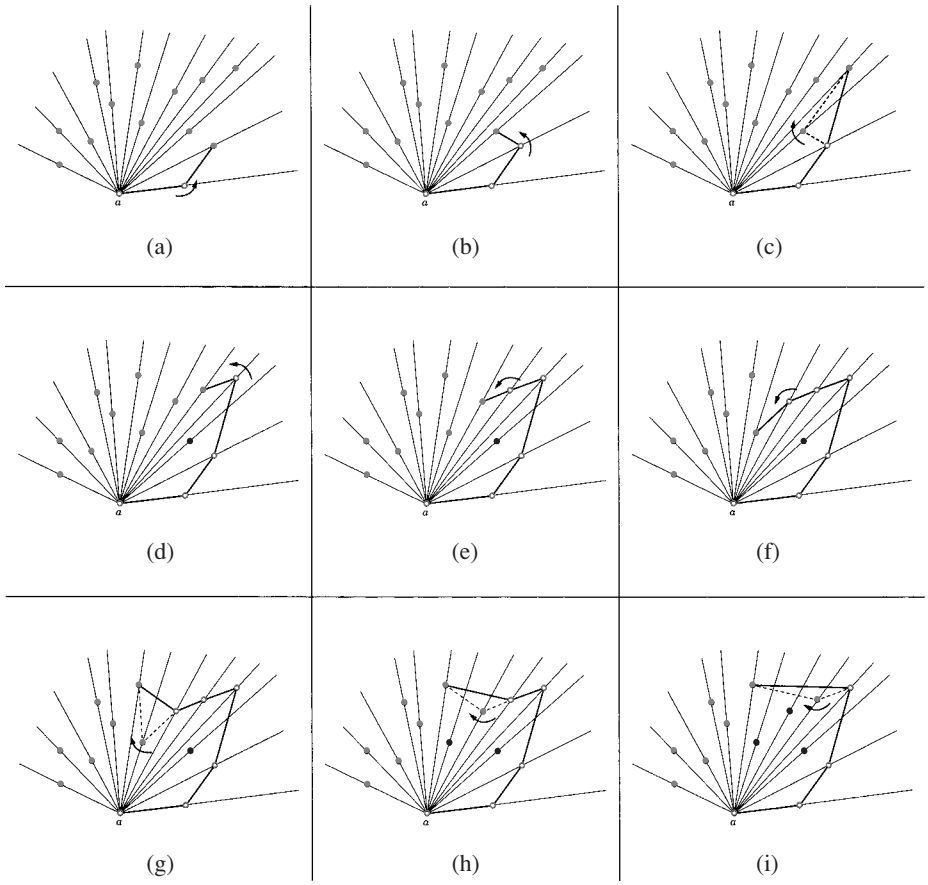


Figura 12.26 Terceira fase do algoritmo de varredura de Graham (veja Algoritmo 12.25).

12.6 Exemplo em Java: cobertura convexa

Descrevemos o algoritmo de varredura de Graham (Algoritmo 12.25) assumindo que ele evita quaisquer ***degenerações***, isto é, configurações de entrada que envolvam casos especiais (tais como pontos coincidentes ou colineares). Quando implementamos o algoritmo, entretanto, é importante lidar com todas as possibilidades de entradas. Por exemplo, dois ou mais pontos serem coincidentes e algumas triplas de pontos serem colineares.

Nos Trechos de Código 12.27 a 12.29, mostramos uma implementação em Java do algoritmo de varredura de Graham. O método principal chama-se grahamScan (Trecho de código 12.27), que usa vários método auxiliares. Em função das configurações degeneradas de pontos que podem ocorrer, várias situações especiais são tratadas pelo método grahamScan.

- A lista de entrada é inicialmente copiada para uma seqüência que será retornada ao final da execução (método copyInputPoints do Trecho de Código 12.28).
- Se a entrada tiver zero ou um ponto (a saída é a mesma que a entrada), retorne.
- Se existirem dois pontos na entrada e esses pontos coincidirem, remova um deles e retorne.
- O ponto de ancoragem é calculado e removido com todos os pontos com ele coincidentes (método anchorPointSearchAndRemove do Trecho de Código 12.28). Se não sobrarem pontos ou sobrar apenas um, reinsira o ponto de ancoragem e retorne.
- Se não ocorrer nenhum dos casos especiais acima, ou seja, existem pelo menos dois pontos sobrando, ordene os pontos no sentido anti-horário ao redor do ponto de ancoragem usando o método sortPoints (Trecho de Código 12.28), que passa um ConvexHullComparator (um comparador que permite a um algoritmo de ordenação genérico ordenar os pontos radialmente no sentido anti-horário ao redor de um ponto, como é necessário para o Algoritmo 12.25).
- Na preparação da varredura de Graham, removemos quaisquer pontos colineares iniciais da lista ordenada, exceto o que estiver mais longe do ponto de ancoragem (método removeInitialIntermediatePoints do Trecho de Código 12.29).
- A fase de varredura do algoritmo é executada chamado o método scan do Trecho de Código 12.29.

Normalmente, quando implementamos algoritmos de geometria computacional, devemos ter um cuidado especial em lidar com todos os casos degenerados.

```
public class ConvexHull {
    private static Sequence hull;
    private static Point2D anchorPoint;
    private static Geom Tester2D geom Tester = new Geom Tester2DImpl();
    // método de classe público
    public static Sequence grahamScan (Sequence points) {
      Point2D p1, p2;
```

```
    copyInputPoints(points);          // copia a seqüência de pontos de entrada
                                      para a cobertura convexa
    switch (hull.size()) {
    case 0: case 1:
      return hull;
    case 2:
      p1 = (Point2D)hull.first().element();
      p2 = (Point2D)hull.last().element();
      if (geom Tester.areEqual(p1,p2))
        hull.remove(hull. last());
      return hull;
    default:     // pelo menos três pontos de entrada
      // calcula ponto de ancoragem e o remove com os pontos
        coincidentes
      anchorPointSearchAndRemove();
      switch (hull.size()) {
      case 0: case 1:
        hull.insertFirst(anchorPoint);
        return hull;
      default:   // pelo menos dois pontos de entrada à esquerda do ponto de
                    ancoragem
        sortPoints(); // ordena os pontos dentro da cobertura convexa ao
                    redor do ponto de ancoragem
        // remove os (possíveis) pontos colineares que estão dentro da
        // cobertura convexa, exceto o que estiver mais longe do ponto de
          ancoragem
        removeInitialIntermediatePoints();
        if (hull.size() == 1)
          hull.insertFirst(anchorPoint);
        else { // insere o ponto de ancoragem como primeiro e último
              elementos da cobertura convexa
          hull.insertFirst(anchorPoint);
          hull. insertLast(anchorPoint);
          scan(); // varredura de Graham
           // remove uma das duas cópias do ponto de ancoragem da cobertura
          convexa
          hull.remove(hull.last());
          }
          return hull;
        }
      }
  }
```

Trecho de Código 12.27 Aplicação do método grahamScan em uma implementação Java do algoritmo de varredura de Graham.

```
    private static void copyInputPoints (Sequence points) {
      // copia a seqüência de pontos de entrada para a cobertura convexa
      hull = new NodeSequence();
      Enumeration pe = points.elements()
      while (pe.hasMoreElements()) {
        Point2D p = (Point2D)pe.nextElement();
        hull. insertLast(p);
    }
}
private static void anchorPointSearchAndRemove () {
    // calcula o ponto de ancoragem e o remove da cobertura convexa juntamente
    // com todos os pontos coincidentes
    Enumeration pe = hull. positions();
    Position anchor = (Position)pe.nextElement();
    anchorPoint = (Point2D)anchor.element();
    // a cobertura convexa contém pelo menos três elementos
    while (pe.hasMoreElements()) {
      Position pos = (Position)pe.nextElement();
      Point2D p = (Point2D)pos.element();
      int aboveBelow = geom Tester.aboveBelow(anchorPoint,p);
      int leftRight = geom Tester.left Right(anchorPoint,p);
      if (aboveBelow == Geom Tester2D.BELOW ||
          aboveBelow == Geom Tester2D.ON &&
          leftRight == Geom Tester2D.LEFT) {
        anchor = pos;
        anchorPoint = p;
      }
      else
        if (aboveBelow == Geom Tester2D.ON &&
            leftRight == Geom Tester2D.ON)
          hull. remove(pos);
    }
    hull. remove(anchor);
}
private static void sortPoints() {
      // ordena os pontos da cobertura convexa ao redor do ponto de ancoragem
      SortObject sorter = new ListMergeSort();
    ConvexHullComparator comp = new ConvexHullComparator(anchorPoint,
                                                          geom Tester);
    sorter.sort(hull,comp);
}
```

Trecho de Código 12.28 Métodos auxiliares copyInputPoints, anchorPointSearchAndRemove e sortPoints usados pelo método grahamScan do Trecho de Código 12.27.

```
private static void removeInitialIntermediatePoints() {
    // remove os (possíveis) pontos colineares iniciais que estão dentro da
    // cobertura convexa, com exceção do mais distante do ponto de
      ancoragem
    boollean collinear = true;
    while (hull.size() > 1&& collinear) {
      Position pos1 = hull.first();
      Position pos2 = hull.after(pos1);
      Point2D p1 = (Point2D)pos1.element();
      Point2D p2 = (Point2D)pos2.element();
      if (geom Tester.leftRightTurn(anchorPoint,p1,p2) ==
          Geom Tester2D.COLLINEAR)
        if (geom Tester.closest(anchorPoint,p1,p2) == p1)
          hull.remove(pos1);
        else
          hull.remove(pos2);
      else
        collinear = false;
    }
}
private static void scan() {
    // varredura de Graham
    Position first = hull.first();
    Position last = hull.last();
    Position prev = hull.first();
    Position curr = hull.after(prev);
    do {
      Position next = hull.after(curr);
      Point2D prevPoint = (Point2D)prev.element();
      Point2D currPoint = (Point2D)curr.element();
      Point2D nextPoint = (Point2D)next.element();
      if (geom Tester.leftRightTurn(prevPoint,currPoint,nextPoint) ==
            Geom Tester2D.LEFT_TURN)
          prev = curr;
        else {
          hull.remove(curr);
          prev = hull.before(prev);
        }
        curr = hull.after(prev);
      }
      while (curr != last);
    }
}
```

Trecho de Código 12.29 Métodos auxiliares removeInitialIntermediatePoints e scan, usados pelo método grahamScan do Trecho de Código 12.27.

12.7 Exercícios

Reforço

R-12.1 Estenda a análise do tempo de execução do Algoritmo 12.1 1DTreeRangeSearch para o caso em que a árvore de pesquisa binária T contém k_1 e/ou k_2.

R-12.2 Verifique se o valor absoluto da função $\Delta(p_1, p_2, p_3)$ corresponde a duas vezes a área do triângulo formado pelos pontos p_1, p_2 e p_3 no plano.

R-12.3 Elabore uma prova completa para o Teorema 12.7.

R-12.4 Elabore uma prova completa para o Teorema 12.8.

R-12.5 Elabore uma prova completa para o Teorema 12.9.

R-12.6 Elabore uma prova completa para o Teorema 12.11.

R-12.7 Elabore uma prova completa para o Teorema 12.12.

R-12.8 Qual seria o pior caso para consumo de espaço de uma árvore intervalar se não for exigido que a estrutura primária tenha altura $O(\log n)$?

R-12.9 Dada um árvore de pesquisa binária T construída a partir das abscissas de um conjunto de n objetos, descreva um método que execute em tempo $O(n)$ para calcular $\text{mín}_x(v)$ e $\text{máx}_x(v)$ para cada nodo v de T.

R-12.10 Demonstre que os valores de high_y em uma árvore de pesquisa com prioridade satisfazem a propriedade de ordenação por *heap*.

R-12.11 Argumente para mostrar que o algoritmo correto para responder uma consulta intervalar de três lados é o que usa uma árvore de pesquisa por prioridade.

R-12.12 Qual é o pior caso em relação à profundidade de uma árvore k-D definida a partir de n pontos sobre o plano? O que ocorre em dimensões maiores?

R-12.13 Suponha um conjunto S que contenha n pontos bidimensionais cujas coordenadas são todas inteiras no intervalo $[0,N]$. Qual o pior caso em relação à profundidade de uma *quadtree* definida sobre S?

R-12.14 Desenhe uma *quadtree* para o seguinte conjunto de pontos, supondo um retângulo limitante de 16×16:

$$\{(1,2), (4,10), (14,3), (6,6), (3,15), (2,2), (3,12), (9,4), (12,14)\}.$$

R-12.15 Construa uma árvore k-D para o conjunto de pontos do Exercício R-12.14.

R-12.16 Construa uma árvore de pesquisa por prioridades para o conjunto de pontos do Exercício R-12.14.

Criatividade

C-12.1 Os rótulos $\text{mín}_x(v)$ e $\text{máx}_x(v)$ usados na árvore intervalar bidimensional não são realmente necessários. Descreva um algoritmo que execute uma consulta intervalar bidimensional em uma árvore intervalar bidimensional onde cada nodo da estrutura primária armazene apenas um rótulo key(v) (que corresponde à abscissa do elemento). Qual o tempo de execução do seu método?

C-12.2 Forneça a descrição em pseudocódigo de um algoritmo para construir uma árvore intervalar a partir de um conjunto de n pontos no plano em tempo $O(n \log n)$.

C-12.3 Descreva uma estrutura de dados eficiente para armazenar um conjunto S de n itens com chaves ordenadas que suportem o método rankRange(a, b) que enumere todos os itens com chaves cuja ***colocação*** em S está no intervalo $[a,b]$, onde a e b são inteiros no intervalo $[0, n-1]$. Descreva métodos para inserção e remoção de objetos e caracterize o tempo de execução para esses métodos e para o rankRange.

C-12.4 Projete uma estrutura de dados estática (que não suporta inserções ou remoções) que armazene um conjunto bidimensional S de n pontos e possa responder a consultas da forma countAllInRange(a, b, c, d), em tempo $O(\log^2 n)$, que retorna o número de pontos em S, com abscissas no intervalo $[a, b]$ e ordenadas no intervalo $[c, d]$. Qual o espaço consumido por essa estrutura?

C-12.5 Projete uma estrutura de dados para responder a consultas do tipo countAllInRange (como definidas no exercício anterior) em tempo $O(\log n)$.

***Dica*:** pense em armazenar em cada nodo estruturas auxiliares que estejam "conectadas" às estruturas dos nodos vizinhos.

C-12.6 Mostre como estender as árvores intervalares bidimensionais de maneira a serem capazes de responder a consultas intervalares d-dimensionais em tempo $O(\log^d n)$ para conjuntos de pontos d-dimensionais, onde $d \geq 2$ é uma constante.

***Dica*:** projete uma estrutura de dados recursiva que construa uma estrutura d-dimensional usando estruturas $(d-1)$ dimensionais.

C-12.7 Suponha uma estrutura de pesquisa intervalar D que possa responder a consultas intervalares para um conjunto de n pontos em um espaço d-dimensional para qualquer dimensão fixa d (como 8, 10 ou 20) em tempo $O(\log^d n + k)$, onde k é o número de respostas. Mostre como usar D para responder às seguintes consultas para um conjunto S de n retângulos no plano:

- findAllContaining(x,y): retorna a enumeração de todos os retângulos de S que contêm o ponto (x,y).
- findAllIntersecting(a, b, c, d): retorna a enumeração de todos os retângulos que interseccionam o retângulo com intervalo no eixo x $[a, b]$ e no eixo y $[c, d]$.

Qual o tempo de execução necessário para responder a essas questões?

C-12.8 Faça S ser um conjunto de n intervalos da forma $[a,b]$, onde $a < b$. Projete uma estrutura de dados eficiente que possa responder em tempo $O(\log n + k)$ a consultas da forma contains(x), que solicitam a enumeração de todos os intervalos de S que contêm x, onde k é a quantidade desses intervalos. Qual o espaço consumido por sua estrutura de dados?

C-12.9 Descreva um método eficiente para fazer inserções em uma árvore de pesquisa balanceada. Qual o tempo de execução deste método?

C-12.10 Suponha uma seqüência S baseada em um arranjo com n segmentos que não se interseccionam $s_0,..., s_{n-1}$ com extremidades nas linhas $y = 0$ e $y = 1$, ordenados da esquerda para a direita. Dado um ponto q com $0 < y(q) < 1$, projete um algoritmo que, em tempo $O(\log n)$, determine o segmento si de S imediatamente à direita de q ou relate que q é o segmento mais à direita de todos.

C-12.11 Suponha um algoritmo que consome um tempo $O(n)$ para testar se um ponto q está dentro, fora ou no limite de um polígono sem intersecções P com n vértices. Seu algoritmo deve trabalhar corretamente, também, no caso em que a coordenada y de q for igual à coordenada y de um ou mais vértices de P.

C-12.12 Projete um algoritmo que consome tempo $O(n)$ para testar quando um polígono de n vértices é convexo. Você não pode assumir que as arestas de P não se interseccionam.

C-12.13 Faça S ser um conjunto de segmentos. Forneça um algoritmo para determinar quando os segmentos de S formam um polígono. Permita que o polígono tenha arestas que se interseccionam; porém, não permita que o mesmo tenha dois vértices coincidentes.

C-12.14 Seja S um conjunto de n segmentos de reta sobre o plano. Forneça um algoritmo para enumerar todos os k pares de segmentos que se interseccionam em S no tempo $O((n + k)\log n)$.

***Dica*:** use a técnica de varredura rotacional, considerando as intersecções como eventos. Observe que você pode não conhecer esses eventos com antecedência, mas sempre é possível saber o próximo evento a processar à medida que a varredura avança.

C-12.15 Projete uma estrutura de dados para armazenar polígonos convexos cuja ocupação de espaço seja linear e suporte o teste de inclusão de ponto em tempo logarítmico.

C-12.16 Dado um conjunto P de n pontos, projete um algoritmo eficiente para construir um polígono sem intersecções cujos vértices são os pontos de P.

C-12.17 Projete um algoritmo que consome tempo $O(n^2)$ para testar se um polígono com n vértices não tem arestas que se interseccionam. Assuma que o polígono é definido pela lista de seus vértices.

C-12.18 Forneça exemplos de configurações de pontos de entrada para os quais o algoritmo simplificado de varredura de Graham, fornecido no Algoritmo 12.25, não funcione corretamente.

C-12.19 Seja P um conjunto de n pontos no plano. Modifique o algoritmo de varredura de Graham para calcular, para cada ponto p de P que não seja vértice da cobertura convexa, tanto um triângulo com vértices em P como um segmento com extremidades em P que contenha p em seu interior.

C-12.20 Dado um conjunto S de pontos no plano, defina o ***diagrama de Voronoi*** de S como o conjunto de regiões $V(p)$, chamadas de ***células de Voronoi***, definidas, para cada ponto de S, como o conjunto de todos os pontos q sobre o plano, tal que p é o vizinho mais próximo de q em S.

a. Mostre que cada célula de um diagrama de Voronoi é convexa.

b. Mostre que, se p e q são o par de pontos mais próximo do conjunto S, então as células de Voronoi $V(p)$ e $V(q)$ se tocam.

c. Mostre que o ponto p está sobre o limite da cobertura convexa do conjunto S se e somente se a célula de Voronoi $V(p)$ para p for não limitada.

C-12.21 Dado um conjunto S de pontos no plano, defina a ***triangulação de Delaunay*** de S como o conjunto de todos os triângulos (p, q, r) tais que p, q e r estejam em S, e o círculo em cuja borda localizam-se esses pontos esteja vazio – não existem pontos de S em seu interior.

a. Mostre que, se p e q são um par de pontos próximos no conjunto S, então p e q estão unidos por uma aresta na triangulação de Delanay.

b. Mostre que as células de Voronoi $V(p)$ e $V(q)$ compartilham uma aresta no diagrama de Voronoi de um conjunto de pontos S se e somente se p e q estiverem unidos por uma aresta na triangulação Delaunay de S.

Projetos

P-12.1 Produza uma animação dos algoritmos do embrulho para presente e de varredura de Graham para o cálculo da cobertura convexa de um conjunto de pontos.

P-12.2 Implemente uma classe que suporte consultas intervalares usando como estrutura de dados árvores intervalares ou árvores intervalares por prioridade.

P-12.3 Implemente as estruturas *quadtree* e árvores k-D e realize um estudo experimental comparando seu desempenho em consultas intervalares.

Notas

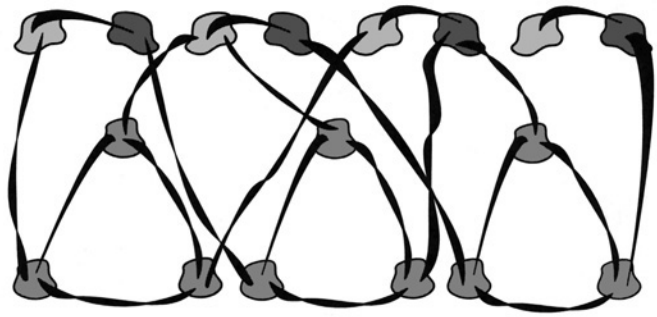
O algoritmo de determinação da cobertura convexa apresentado neste capítulo é uma variante do algoritmo apresentado por Graham [88]. O algoritmo do plano rotacional apresentado para a determinação de segmentos de linha ortogonais que se interseccionam é atribuído a Bentley e Ottman [31]. O algoritmo de determinação do ponto mais próximo combina idéias de Bentley [28] e Hinrichs *et al*. [94].

Existem vários livros excelentes sobre geometria computacional, incluindo os livros de Edelsbrunner [63], Mehlhorn [150], O'Rourke [160], Preparata e Shamos [168] e manuais editados por Goodman e O'Rourke [83] e Pach [162]. Outras fontes de leitura adicionais incluem os artigos de revisão de Aurenhammer [17], Lee e Preparata [129] e capítulos de livros de Goodrich [84], Lee [128] e Yao [212]. Além disso, os livros de Sedgewick [182, 183] contêm vários capítulos sobre geometria computacional, com ótimas figuras. Na verdade, as figuras do livro de Sedgewick inspiraram muitas das figuras que mostramos neste livro.

Árvores de pesquisa multidimensionais são discutidas nos livros de Mehlhorn [150], Samet [175,176] e Wood [211]. Consulte esses livros para uma discussão mais profunda da história da pesquisa sobre árvores de pesquisa multidimensionais, incluindo várias estruturas de dados para resolver consultas intervalares. Árvores de pesquisa por prioridade são atribuídas a McCreight [140], embora Vuillemin [208] tenha introduzido essa estrutura mais tarde sob o nome de "árvores cartesianas". Elas também são conhecidas como *treaps*, como descritas por McCreight [140] e Aragon e Seidel [12]. Edelsbrunner [62] mostra de que forma as árvores por prioridade podem ser usadas para responder a consultas intervalares bidimensionais. Arya e Mount [14] apresentamas árvores de decomposição por caixas balanceadas e mostram como elas podem ser usadas para resolver uma pesquisa intervalar por aproximação [15]. Ao leitor interessado nos estudos mais recentes sobre estruturas de dados para pesquisa intervalar, indicamos os capítulos de livro de Agarwal [3,4] ou o artigo de Matousêk [138]. Luca Vismara desenvolveu a implementação do algoritmo de cobertura convexa apresentado na Seção 12.6.

Capítulo

13 NP-completude



13.1 P e NP 584

13.1.1 Definição das classes de complexidade **P** e **NP** 585

13.1.2 Alguns problemas interessantes em **NP** 588

13.2 NP-completude 590

13.2.1 Redutibilidade em tempo polinomial e **NP**-dificuldade 590

13.2.2 O teorema de Cook-Levin 591

13.3 Problemas NP-completos importantes 593

13.3.1 CNF-SAT e 3SAT 595

13.3.2 VERTEX-COVER 599

13.3.3 CLIQUE e SET-COVER 601

13.3.4 SUBSET-SUM e KNAPSACK 602

13.3.5 CICLO HAMILTONIANO e TSP 606

13.4 Algoritmos aproximativos 608

13.4.1 Esquemas de aproximação de tempo polinomial 609

13.4.2 Uma 2-aproximação para VERTEX-COVER 612

13.4.3 Uma 2-aproximação para um caso especial de TSP 613

13.4.4 Uma aproximação logarítmica para SET-COVER 614

13.5 *Backtracking* e *branch-and-bound* 616

13.5.1 *Backtracking* 617

13.5.2 *Branch-and-bound* 621

13.6 Exercícios 627

Alguns problemas computacionais são difíceis. Tentamos e tentamos encontrar algoritmos eficientes para resolvê-los, mas falhamos repetidamente. Seria bom se pudéssemos provar que é impossível encontrar um algoritmo eficiente nestes casos. Tal prova seria útil quando não achamos um algoritmo eficiente, pois poderíamos nos tranqüilizar sabendo que um algoritmo desse tipo não pode ser achado. Infelizmente esse tipo de prova é tão difícil de obter quanto os algoritmos eficientes.

O cenário não é tão frustrante, pois os tópicos que discutiremos neste capítulo mostram que certos problemas são mesmo computacionalmente difíceis. As provas envolvem o conceito de ***NP-completude***, que permite que mostremos rigorosamente que encontrar um algoritmo eficiente para um certo problema é pelo menos tão difícil quanto encontrar algoritmos eficientes para ***todos*** problemas em uma grande classe de problemas chamada de ***NP***. A noção formal de "eficiente" que usaremos significa que um problema tem um algoritmo executado em tempo proporcional a uma função polinomial de seu tamanho de entrada n (lembre que essa noção de eficiência já foi mencionada na Seção 1.2.2). Ou seja, consideramos um algoritmo "eficiente" se ele for executado em tempo $O(n^k)$ para alguma constante $k > 0$. Mesmo assim, a classe ***NP*** contém alguns problemas extremamente difíceis, para os quais as soluções de tempo polinomial têm escapado dos pesquisadores durante anos. Portanto, embora mostrar que um problema é ***NP***-completo não seja a mesma coisa que mostrar que um algoritmo eficiente para o problema é impossível, ainda assim é uma afirmação bastante forte. Basicamente, mostrar que um problema L é ***NP***-completo diz que, embora não sejamos capazes de achar um algoritmo eficiente para L, nenhum outro cientista pode ter conseguido. De fato, a maior parte dos cientistas da computação acredita firmemente que é impossível resolver qualquer problema ***NP***-completo em tempo polinomial.

Neste capítulo, definimos formalmente a classe ***NP*** e sua classe associada ***P***, e mostramos como provar que alguns problemas são ***NP***-completos. Também discutimos alguns dos mais conhecidos problemas ***NP***-completos, mostrando que cada um é pelo menos tão difícil quanto qualquer outro problema em ***NP***. Esses problemas incluem satisfatibilidade, cobertura de vértices, o problema da mochila e o do caixeiro viajante.

Não terminaremos aí, pois muitos desses problemas são importantes e estão relacionados a problemas de otimização cuja solução no mundo real pode economizar tempo, dinheiro ou outros recursos. Por isso, discutiremos maneiras de lidar com ***NP***-completude neste capítulo. Um dos métodos mais efetivos é construir um algoritmo de aproximação de tempo polinomial para problemas ***NP***-completos. Embora esses algoritmos geralmente não produzam soluções ótimas, eles com freqüência chegam perto dela. Na verdade, em alguns casos, podemos oferecer uma garantia de quão perto o algoritmo aproximativo irá chegar de uma solução ótima. Exploraremos várias dessas situações neste capítulo.

Concluímos o capítulo com técnicas que costumam funcionar bem com problemas ***NP***-completos na prática. Apresentaremos as técnicas de ***backtracking*** e ***branch-and-bound***, que constroem algoritmos que são executados em tempo exponencial no pior caso, mas tiram vantagem de situações em que um tempo menor pode ser atingido. Forneceremos exemplos em Java de ambas as técnicas.

13.1 P e NP

Para estudar a ***NP***-completude, precisamos ser mais precisos sobre o tempo de execução. Em vez da noção informal do tamanho da entrada como o número de "itens" que formam a entrada (veja o Capítulo 1), definimos o ***tamanho da entrada*** n de um problema como o número de *bits* usados para codificar uma instância do problema. Também assumimos que os números e caracteres da entrada são codificados usando um esquema razoável de codificação binária, em que cada caracter usa um número constante de *bits* e cada inteiro $M > 0$ é representado com no máximo $c \log M$ *bits* para alguma constante $c > 0$. Em particular, não permitimos a ***codificação unária***, em que um inteiro M é representado com M valores 1.

Lembre que, no restante deste livro, definimos o tamanho da entrada como o número de "itens" que formam a entrada. No entanto, vamos por um momento falar do número de itens na entrada como N, e do número de *bits* para codificar a entrada como n. Assim, se M for o maior inteiro na entrada, então $N + \log M \leq n \leq cN \log M$, para alguma constante $c > 0$. Formalmente, definimos o ***tempo de execução*** de pior caso do algoritmo A como o pior tempo que A pode ter em função de n, que é tomado sobre todas as entradas que têm uma codificação de n *bits*. Felizmente, como mostraremos no lema a seguir, muitos algoritmos que rodam em tempo polinomial em função de N também rodam em tempo polinomial em função de n. Definimos um algoritmo como ***c-incremental*** se qualquer operação primitiva envolvendo um ou dois objetos representados com b *bits* resultar em um objeto representado com no máximo $b + c$ *bits*, com $c \geq 0$. Por exemplo, um algoritmo usando a multiplicação como operação primitiva pode não ser c-incremental para nenhuma constante c. É claro que podemos ter uma rotina de multiplicação em um algoritmo c-incremental, mas não devemos contabilizá-la como uma operação primitiva.

Lema 13.1: *se um algoritmo c-incremental A tiver tempo de execução no pior caso $t(N)$ no modelo RAM, como uma função do número de itens de entrada N para alguma constante $c > 0$, então A tem tempo de execução $O(n^2 t(n))$ em função do número n de bits em uma codificação não-unária da entrada.*

Prova: note que $N \leq n$. Assim, $t(N) \leq t(n)$. De forma similar, cada operação primitiva no algoritmo A envolvendo um ou dois objetos representados com $b \geq 1$ *bits* pode ser realizada usando no máximo db^2 operações com relação a *bit*, para alguma constante $d \geq 1$, pois c é uma constante. Tais operações primitivas incluem comparações, controle de fluxo e operações aritméticas básicas e não-multiplicativas. Além disso, em N etapas de um algoritmo c-incremental, a maior representação que um objeto pode atingir é $cN + b$, onde b é o maior tamanho de um objeto da entrada. Mas $cN + b \leq (c + 1)n$. Assim, cada etapa em A custará no máximo $O(n^2)$ operações entre *bits* para ser completado. ■

Assim, qualquer algoritmo "razoável" de tempo polinomial em termos do número de itens de entrada também será de tempo polinomial em termos do número de *bits* de entrada. Assim, ao longo deste capítulo, podemos voltar a usar n como tamanho de entrada e número de itens, com o entendimento de que qualquer algoritmo de "tempo polinomial" deve ter tempo polinomial em função do número de *bits* de entrada.

13.1.1 Definição das classes de complexidade **P** e **NP**

Pelo Lema 13.1, sabemos que, para os problemas discutidos neste livro como problemas de grafos, processamento de texto e ordenação, nossos algoritmos anteriores de tempo polinomial se traduzem em algoritmos de tempo polinomial em relação ao número de *bits*. Mesmo o algoritmo de quadrados sucessivos (Seção 10.1.4) para calcular potências de um inteiro x tem um número polinomial de operações entre *bits* quando aplicado para elevar x a uma potência representada usando $O(\log(n))$ *bits*. Assim, a noção de tempo polinomial é útil como medida de tratabilidade.

Além disso, a classe dos polinômios é fechada sob adição, multiplicação e composição. Ou seja, se $p(n)$ e $q(n)$ são polinômios, então $p(n) + q(n)$, $p(n) \cdot q(n)$ e $p(q(n))$ também são. Assim, podemos combinar ou compor algoritmos de tempo polinomial para obter novos algoritmos de tempo polinomial.

Problemas de decisão

Para simplificar a discussão, vamos restringir nossa atenção no momento aos ***problemas de decisão***, ou seja, problemas computacionais cuja saída esperada é apenas "sim" ou "não". Em outras palavras, a saída de um problema de decisão é um único *bit,* que é 0 ou 1. Por exemplo, os problemas abaixo são problemas de decisão:

- Dadas duas cadeias de caracteres T e P, P aparece como uma subseqüência de T?
- Dados dois conjuntos S e T, eles contêm os mesmos elementos?
- Dado um grafo G com pesos inteiros nas arestas e um inteiro k, G tem uma árvore de cobertura mínima com peso não maior do que k?

Na verdade, o último problema ilustra como podemos transformar um ***problema de otimização***, em que desejamos maximizar ou minimizar algum valor, em um problema de decisão. Para isso, introduzimos um parâmetro k e perguntamos se o valor ótimo para o problema de otimização é no máximo ou no mínimo k. Note que, se pudermos mostrar que um problema de decisão é difícil, então sua versão de otimização também deve ser difícil.

Problemas e linguagens

Dizemos que um algoritmo A ***aceita*** uma entrada x se A responder "sim" para a entrada x. Por isso, podemos ver um ***problema de decisão*** como um conjunto L de entradas — as entradas que devem ser aceitas por um algoritmo que resolve o problema. Usamos a letra L para denotar um problema de decisão porque um conjunto de entradas é às vezes chamado de ***linguagem***. Podemos estender esta visão baseada em linguagens e dizer que um algoritmo A ***aceita*** uma linguagem L se A responder "sim" para todo x em L e "não" em outros casos. Neste capítulo, assumiremos que, se x tiver um formato impróprio, o algoritmo responderá "não". (Nota: alguns textos também permitem que A entre em laço infinito para algumas entradas e nunca forneça uma saída, mas restringimos nossa atenção neste livro nos algoritmos que terminam após um número finito de passos.)

A classe de complexidade *P*

A ***classe de complexidade P*** é o conjunto de todos os problemas de decisão (ou linguagens) L que podem ser aceitos em tempo polinomial no pior caso. Ou seja, existe um algoritmo A que, dado $x \in L$, responde "sim" em tempo $p(n)$, onde n é o tamanho de x e $p(n)$ é um polinômio. Note que a definição de ***P*** não diz nada sobre o tempo de execução para rejeitar uma entrada — quando o algoritmo A responde "não". Tais casos representam o ***complemento*** da linguagem L, que consiste em todas as seqüências binárias que não estão em L. Dado um algoritmo A que aceita uma linguagem L em tempo polinomial $p(n)$, podemos facilmente construir um algoritmo de tempo polinomial que aceita o complemento de L. Em particular, dada uma entrada x, podemos construir um algoritmo B para o complemento que simplesmente execute A por $p(n)$ etapas, onde n é o tamanho da entrada x, encerrando A se ele tentar rodar por mais de $p(n)$ etapas. Se A responder "sim", então B responde "não". Da mesma forma, se A responder "não" ou se A tentar rodar por mais de $p(n)$ etapas sem responder nada, então B responde "sim". Em qualquer caso, o algoritmo B para o complemento roda em tempo polinomial. Portanto, se uma linguagem L representando um problema de decisão estiver em ***P***, então o complemento de L também está em ***P***.

A classe de complexidade *NP*

A ***classe de complexidade NP*** é definida incluindo a classe de complexidade ***P*** permitindo, entretanto, a inclusão de linguagens que podem não estar em ***P***. Especificamente com problemas ***NP***, permitimos que algoritmos realizem uma operação adicional:

- choose(b): esta operação escolhe de forma não-determinística um *bit* (ou seja, um valor 0 ou 1) e este é atribuído a b.

Quando um algoritmo A usa a operação primitiva *choose*, dizemos que A é ***não-determinístico***. Afirmamos que um algoritmo A ***aceita deterministicamente*** uma entrada x se existir um conjunto de saídas da operação choose que A poderia usar para responder "sim". Em outras palavras, é como se considerássemos todas as saídas possíveis de choose e escolhêssemos apenas as que levam à aceitação, se existir um tal conjunto de saídas. Note que isso não é o mesmo que escolhas aleatórias.

A classe de complexidade ***NP*** é o conjunto de todos os problemas de decisão (ou linguagens) L que podem ser aceitos não-deterministicamente em tempo polinomial. Ou seja, se existe um algoritmo não-determinístico A que, dado $x \in L$, então existe um conjunto de saídas da função choose em A tal que A responde "sim" em tempo $p(n)$, onde n é o tamanho de x e $p(n)$ é um polinômio.

Note que a definição de ***NP*** não diz nada sobre o tempo de execução para rejeitar uma entrada. Admitimos que um algoritmo A que aceita uma linguagem L em tempo polinomial $p(n)$ use muito mais do que $p(n)$ etapas quando A responde "não". Além disso, como a aceitação não-determinística pode envolver um número polinomial de chamadas da função choose, se uma linguagem L estiver em ***NP***, então o complemento de L não está necessariamente em ***NP***. Na verdade, existe uma classe de complexidade chamada **co-*NP*** que consiste em todas as linguagens cujo complemento está em ***NP***, e muitos pesquisadores acreditam que **co-*NP*** $\neq$ ***NP***.

Uma definição alternativa de ***NP***

Existe outra maneira de definir a classe de complexidade ***NP*** que pode ser mais intuitiva para alguns leitores. Essa definição alternativa de ***NP*** é baseada na verificação determinística, em vez de não-determinística. Dizemos que uma linguagem L pode ser ***verificada*** por um algoritmo A se, dada uma entrada x de L, existir outra entrada y tal que A responda "sim" para a entrada $z = x + y$, onde usamos o símbolo '+' para representar concatenação. A entrada y é chamada de ***certificado*** para L, pois ela ajuda a assegurar que x pertence a L. Note que não afirmamos nada sobre a verificação se a entrada não estiver em L.

A noção de verificação permite uma definição alternativa da classe de complexidade ***NP***. Podemos definir ***NP*** como o conjunto das linguagens L que definem problemas de decisão tais que L pode ser verificada em tempo polinomial. Ou seja, existe um algoritmo (determinístico) A que, para qualquer x em L, verifica em tempo polinomial $p(n)$ e usando um certificado y que x está em L. Esse tempo inclui o tempo que A leva para ler sua entrada $z = x + y$, onde n é o tamanho de x. Note que essa definição implica que o tamanho de y seja menor do que $p(n)$. Como o teorema a seguir mostra, essa definição de ***NP*** baseada em verificação é equivalente à definição baseada em não-determinismo apresentada acima.

Teorema 13.2: *uma linguagem L pode ser* (*deterministicamente*) *verificada em tempo polinomial se e somente se L puder ser não-deterministicamente aceita em tempo polinomial.*

Prova: vamos considerar cada possibilidade. Supomos primeiro que L possa ser verificada em tempo polinomial, ou seja, existe um algoritmo determinístico A (sem usar a função choose) que pode verificar em tempo polinomial $p(n)$ que uma entrada x está em L quando for fornecido um certificado y de tamanho polinomial. Portanto, podemos construir um algoritmo não-determinístico B que receba x como entrada e use o método choose para definir o valor de cada *bit* em y. Depois que B constrói uma entrada $z = x + y$, ele então usa A para verificar que $x \in L$, dado o certificado y. Se existir um certificado y tal que A aceita z, então existe um conjunto de escolhas não-determinísticas para B que resultariam na saída "sim". Além disso, B é executado em $O(p(n))$ etapas.

Em seguida, suponha que L possa ser aceita não-deterministicamente em tempo polinomial, ou seja, existe um algoritmo não-determinístico A que, dada uma entrada x em L, realiza $p(n)$ etapas que podem incluir avaliações da função choose tais que, para alguma seqüência de valores dados pela função choose, A responda "sim". Existe um algoritmo determinístico de verificação B que, dada uma entrada x em L, usa como certificado y a concatenação ordenada de todos resultados de choose que A pode fazer para a entrada x até responder "sim". Como A é executado em $p(n)$ etapas, onde n é o tamanho de x, o algoritmo B também é executado em $O(p(n))$ etapas para a entrada $z = x + y$. ■

A implicação prática desse teorema é que, como as duas definições de ***NP*** são equivalentes, podemos usar qualquer uma delas para mostrar que um problema está em ***NP***.

A questão ***P* = *NP***

Os cientistas da computação não sabem ao certo se ***P* = *NP*** ou não. De fato, os pesquisadores não sabem nem mesmo se ***P* = *NP*** ∩ **co-*NP***. Mesmo assim, a maioria dos pesquisadores acredita que ***P*** seja diferente de ***NP*** e de **co-*NP***, bem como seja diferente de sua intersecção. Na realidade, os problemas que discutiremos a seguir são exemplos de problemas que pertencem a ***NP*** e que acredita-se não pertencerem a ***P***.

13.1.2 Alguns problemas interessantes em *NP*

Outra interpretação do Teorema 13.2 é que ele implica que podemos sempre estruturar um algoritmo não-determinístico de forma que as avaliações de choose sejam realizadas primeiramente e, que o resto do algoritmo seja apenas uma verificação. Ilustramos esta abordagem com vários exemplos nesta subseção, mostrando que alguns problemas de decisão interessantes estão em ***NP***. Nosso primeiro problema está relacionado a grafos.

O CICLO-HAMILTONIANO é o problema que recebe um grafo G como entrada e pergunta se existe um ciclo simples em G que visite cada vértice de G exatamente uma vez e retorne ao vértice inicial. Um ciclo desse tipo é chamado de ciclo hamiltoniano de G.

Lema 13.3: *O* CICLO-HAMILTONIANO *está em* ***NP***.

Prova: vamos definir um algoritmo não-determinístico A que receba como entrada um grafo G codificado como lista de adjacência em notação binária com os vértices numerados de 1 a N. Definimos que A chama inicialmente a função choose para determinar a seqüência S de $N + 1$ números entre 1 e N. Em seguida, A verifica que cada número de 1 a N aparece exatamente uma vez em S (por exemplo, ordenando S), exceto pelo primeiro e último números em S, que devem ser iguais. Então verificamos que a seqüência S define um ciclo de vértices e arestas em G. Uma codificação binária da seqüência S tem tamanho máximo n, onde n é o tamanho da entrada. Além disso, as verificações feitas na seqüência S podem ser feitas em tempo polinomial em n. ■

Observe que, se existe um ciclo em G que visita cada vértice de G exatamente uma vez e retorna ao vértice inicial, então existe uma seqüência S para a qual A responde "sim". Da mesma forma, se A responde "sim", então foi encontrado um ciclo em G que visita cada vértice de G exatamente uma vez e retorna ao vértice inicial. Ou seja, A aceita não-deterministicamente a linguagem CICLO-HAMILTONIANO. Em outras palavras, o CICLO-HAMILTONIANO está em ***NP***.

O próximo exemplo é um problema relacionado ao teste de circuitos. Um ***circuito booleano*** é um grafo dirigido em que cada nodo, chamado de ***porta lógica***, corresponde a uma função booleana simples, AND, OR ou NOT. As arestas de entrada em uma porta lógica correspondem às entradas para sua função booleana, e as arestas de saída correspondem à saída da função (veja a Figura 13.1). Os vértices sem arestas de entrada são chamados de ***entradas***, e um vértice sem arestas de saída é chamado de ***saída***.

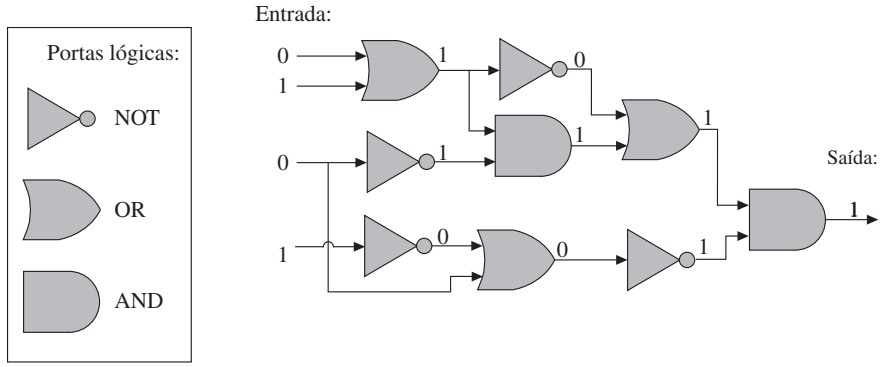


Figura 13.1 Um exemplo de circuito booleano.

O CIRCUIT-SAT é o problema que recebe como entrada um circuito booleano com um único nodo de saída e pergunta se existe uma atribuição de valores para as entradas do circuito de forma que a saída seja "1". Uma atribuição de valores desse tipo é chamada de ***atribuição satisfatória***.

Lema 13.4: *O* CIRCUIT-SAT *está em* ***NP***.

Prova: construímos um algoritmo não-determinístico para aceitar CIRCUIT-SAT em tempo polinomial. Primeiramente, usamos o método choose para "adivinhar" valores para as entradas e valores de saída de cada porta lógica. Então visitamos cada porta lógica g em G, ou seja, cada vértice com pelo menos uma aresta de entrada, e verificamos se o valor adivinhado para a saída de g é o valor de saída correto da função booleana de g (AND, OR e NOT), de acordo com as entradas para g. Esse processo de avaliação pode ser realizado facilmente em tempo polinomial. Se a verificação para um nodo g falhar, ou se o valor adivinhado para a saída for 0, então respondemos "não". Se o teste de cada nodo for bem sucedido e o valor da saída for 1, então o algoritmo responde "sim". Desta forma, se existir uma atribuição satisfatória de valores de entrada para G, então existe uma possível coleção de saídas de choose, de forma que o algoritmo responde "sim" em tempo polinomial. Da mesma forma, se existir uma possível coleção de saídas de choose de forma que o algoritmo responda "sim" em tempo polinomial, então existe uma atribuição satisfatória de valores de entrada para G. Portanto, o CIRCUIT-SAT está em ***NP***. ■

O próximo exemplo ilustra como uma versão de decisão de um problema de otimização pode ser associada a ***NP***. Dado um grafo G, uma ***cobertura*** de G é um subconjunto C de vértices tais que, para cada aresta (v, w) de G, temos $v \in C$ ou $w \in C$ (possivelmente ambos). A meta da otimização é encontrar uma cobertura de G que seja a menor possível.

A COBERTURA é o problema de decisão que recebe um grafo G e um inteiro k como entrada e determina se existe uma cobertura para G contendo no máximo k vértices.

Lema 13.5: *A* COBERTURA *está em* ***NP***.

Prova: suponha que recebemos um inteiro k e um grafo G com os vértices de G numerados de 1 a N. Podemos usar o método choose para criar uma coleção C de k números na faixa de 1 a N. Como verificação, inserimos todos os números de C em um dicionário e examinamos cada uma das arestas de G para verificar que, para cada aresta (v, w) em G, temos que v está em C ou w está em C. Se encontrarmos uma aresta que não tenha nenhum de seus vértices em C, então respondemos "não". Se passarmos por todas as arestas de G e cada uma tiver ao menos um vértice em C, respondemos "sim". Essa verificação claramente consome tempo polinomial.

Note que se G tem uma cobertura de tamanho menor ou igual a k, então existe uma atribuição de números que define a coleção C de forma que cada aresta de G passa no teste e nosso algoritmo responde "sim". De forma similar, se nosso algoritmo responde "sim" então deve haver um subconjunto C de tamanho menor ou igual a k de vértices de G tais que C é uma cobertura para G. Assim, a COBERTURA está em ***NP***. ■

Depois de comentar exemplos de problemas em ***NP***, vamos agora tratar da definição de ***NP***-completude.

13.2 *NP*-completude

A noção de aceitação não-determinística de um problema de decisão (ou linguagem) é admitidamente estranha. Afinal, não existe um computador convencional que possa executar eficientemente um algoritmo não-determinístico com muitas chamadas para choose. De fato, até hoje ninguém mostrou nem como um computador não-convencional, tal qual um computador quântico ou de DNA, pode simular eficientemente um algoritmo não-determinístico de tempo polinomial usando um total polinomial de recursos. Certamente podemos simular deterministicamente um algoritmo não-determinístico examinando todas as saídas possíveis de choose que o algoritmo pode fazer. Esta simulação, entretanto, levaria tempo exponencial para qualquer algoritmo não-determinístico que faça ao menos n^{ε} chamadas a choose, para qualquer constante $\varepsilon > 0$. De fato, existem centenas de problemas na classe de complexidade ***NP*** para os quais a maior parte dos cientistas da computação acredita fortemente que nenhum método determinístico convencional possa resolver em tempo polinomial.

A utilidade da classe de complexidade ***NP*** é que ela captura formalmente vários problemas que muitos acreditam ser computacionalmente difíceis. Na verdade, existem problemas que podem ser provados como tão difíceis quanto qualquer outro problema em ***NP*** no que diz respeito a soluções de tempo polinomial. Essa noção de dificuldade é baseada no conceito de redutibilidade em tempo polinomial, que discutiremos a seguir.

13.2.1 Redutibilidade em tempo polinomial e *NP*-dificuldade

Dizemos que a linguagem L que define um problema de decisão é ***redutível em tempo polinomial*** a uma linguagem M se existir uma função f computável em tempo polinomial que recebe uma entrada x para L e a transforma em uma entrada $f(x)$ para M tal que $x \in L$ se e somente se $f(x) \in M$. Além disso, usamos uma notação abreviada $L \xrightarrow{\text{poli}} M$ para indicar que a linguagem L é redutível em tempo polinomial à linguagem M.

Dizemos que uma linguagem M que define um problema de decisão é ***NP-difícil*** se cada outra linguagem L em **NP** for redutível em tempo polinomial a M. Em notação mais matemática, M é ***NP***-difícil se, para cada $L \in$ ***NP*** tivermos $L \xrightarrow{\text{poli}} M$. Se uma linguagem M for ***NP***-difícil e também estiver na classe ***NP***, então M é ***NP***-completa. Assim, um problema ***NP***-completo é, em um sentido bastante formal, um dos mais difíceis problemas em ***NP*** no que diz respeito à computabilidade em tempo polinomial. Por isso, se alguém provar que um problema ***NP***-completo L pode ser resolvido em tempo polinomial, então isto imediatamente implicará que qualquer outro problema em toda a classe ***NP*** pode ser resolvido em tempo polinomial. Neste caso, poderíamos aceitar qualquer linguagem M em ***NP*** reduzindo-a a L e executando o algoritmo para L. Em outras palavras, se alguém encontrar um algoritmo determinístico de tempo polinomial para apenas um problema ***NP***-completo, então ***P = NP***.

13.2.2 O teorema de Cook-Levin

A princípio pode parecer que a definição de ***NP***-completude é muito forte. Mesmo assim, como o teorema a seguir mostra, existe ao menos um problema ***NP***-completo.

Teorema 13.6 (teorema de Cook-Levin): CIRCUIT-SAT *é* ***NP****-completo*.

Prova: o Lema 13.4 mostra que CIRCUIT-SAT está em ***NP***. Assim, temos de mostrar que o problema é ***NP***-completo, ou seja, temos de mostrar que cada problema em ***NP*** é redutível em tempo polinomial a CIRCUIT-SAT. Então, considere uma linguagem L representando um problema de decisão que está em ***NP***. Como L está em ***NP***, existe um algoritmo determinístico D que aceita qualquer x em L em tempo polinomial $p(n)$ dado um certificado y de tamanho polinomial onde n é o tamanho de x. A idéia da prova é construir um circuito C, possivelmente grande mas de tamanho polinomial, que simule o algoritmo D em uma entrada x de forma que C possa ser satisfeito se e somente se existir um certificado y tal que D responda "sim" para a entrada $z = x + y$.

Lembre (da Seção 1.1.2) que qualquer algoritmo determinístico como D pode ser implementado em um modelo computacional simples (chamado de Máquina de Acesso Aleatório, ou RAM*) que consiste em uma CPU e um banco M de posições de memória com endereços. Em nosso caso, a memória M contém a entrada x, o certificado y, a área de trabalho W que D precisa para realizar sua computação e o código para o próprio algoritmo D. A área de trabalho W de D inclui todos os registradores usados para cálculos temporários e procedimentos usados por D durante sua execução. Entre outros dados, a área W contém um contador de programa (PC) que identifica que instrução D está executando a cada momento. Assim, não existem posições de memória na CPU, mas ao realizar cada etapa de D, a CPU faz a leitura da próxima instrução i apontada pelo PC, efetua a operação indicada por i, seja ela uma comparação, operação aritmética ou desvio, e atualiza o PC para apontar para a próxima instrução a ser realizada. Com isso, o estado corrente de D é completamente caracterizado pelo conteúdo da área de memória M. Além disso, como D aceita x em L em um número de etapas polinomial $p(n)$ onde n é o tamanho de x, então toda a área de memória pode ser considerada como um conjunto de $p(n)$ *bits*. Isso acontece porque, em $p(n)$ etapas, D

* N. de R. Do inglês, *Random Access Machine*.

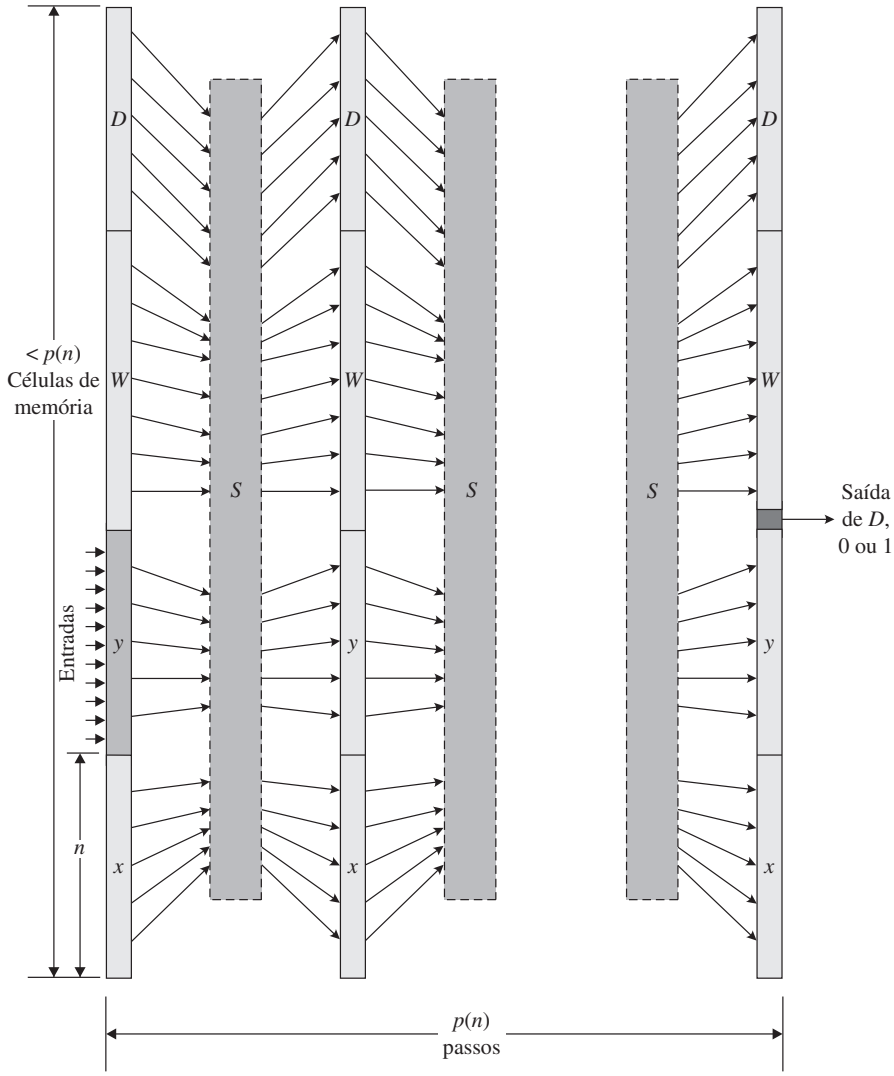


Figura 13.2 Ilustração do circuito usado para provar que CIRCUIT-SAT é ***NP***-completo. As únicas entradas verdadeiras são as que correspondem ao certificado y. A instância do problema x, a área de trabalho W e o código do programa D são inicialmente predeterminados. A única saída é o *bit* que determina se o algoritmo aceita x ou não.

pode acessar no máximo $p(n)$ posições de memória. Note também que o tamanho do código de D é constante em relação aos tamanhos de x, y e W. Falaremos da coleção M composta de $p(n)$ posições de memória como a ***configuração*** do algoritmo D.

A base da redução de L para CIRCUIT-SAT depende da construção de um circuito booleano que simula o funcionamento da CPU em nosso modelo computacional. Os detalhes de uma construção desse tipo estão além do escopo deste livro, mas é sabido que uma CPU pode ser construída como um circuito booleano que consiste nas portas AND, OR e NOT. Além disso, vamos supor que esse circuito, incluindo sua unidade de endereçamento para conexão com a memória de $p(n)$ *bits*, possa ser construído para receber a configuração de D como entrada e fornecer como saída a configuração resultante do processamento da próxima instrução. Assim, esse circuito de simulação que chamaremos de S pode ser construído consistindo em no máximo $cp(n)^2$ portas AND, OR e NOT, para alguma constante $c > 0$.

Para simular as $p(n)$ etapas de D, fazemos $p(n)$ cópias de S com a saída de uma cópia servindo de entrada para a seguinte (veja a Figura 13.2). Parte da entrada para a primeira cópia de S consiste em valores predeterminados para o programa de D, no valor de x, no valor inicial do PC apontando para a primeira instrução de D e no restante da área de trabalho W (inicializada com zeros). A única entrada não-especificada para a primeira cópia de S são as células da configuração de D para o certificado y. Essas são as verdadeiras entradas para o circuito. Da mesma forma, ignoramos todas as saídas da última cópia de S, a não ser a saída que indica se a resposta de D é 1 ("sim") ou 0 ("não"). O tamanho total do circuito C é $O(p(n)^3)$, que é polinomial no tamanho de x.

Considere uma entrada x que aceita D para algum certificado y após $p(n)$ etapas. Então existe uma atribuição de valores da entrada para C correspondendo a y, de forma que se C simular D para esta entrada e para os valores predeterminados para x, então teremos 1 como saída de C. Assim, C pode ser satisfeito neste caso. Da mesma forma, considere um caso em que C possa ser satisfeito. Então existe um conjunto de entradas que correspondem a y para o qual C fornece 1 como resposta. Mas como C simula exatamente o algoritmo D, isso implica que existe uma atribuição de valores para o certificado y tal que D fornece "sim" como resposta. Então, D irá verificar x neste caso. Portanto, D aceita x com certificado y se e somente se C puder ser satisfeito. ■

13.3 Problemas *NP*-completos importantes

Existe de fato ao menos um problema ***NP***-completo. Por outro lado, provar esse fato foi um exercício cansativo mesmo levando em conta o atalho que tomamos ao assumir que havia um circuito de simulação S. Felizmente, agora que temos um problema que já foi comprovado como ***NP***-completo desde o início, podemos provar que outros problemas são ***NP***-completos usando reduções de tempo polinomial. Exploraremos algumas dessas reduções nesta seção.

Dado um problema ***NP***-completo, podemos agora usar a redutibilidade em tempo polinomial para mostrar que outros problemas são ***NP***-completos. Além disso, faremos uso por várias vezes do importante lema a seguir, que fala da redutibilidade em tempo polinomial.

Lema 13.7: *se* $L_1 \xrightarrow{\text{poli}} L_2$ *e* $L_2 \xrightarrow{\text{poli}} L_3$, *então* $L_1 \xrightarrow{\text{poli}} L_3$.

Prova: como $L_1 \xrightarrow{\text{poli}} L_2$, qualquer instância x para L_1 pode ser convertida em tempo polinomial $p(n)$ em uma instância $f(x)$ para L_2, tal que $x \in L_1$ se e somente se $x \in L_2$, onde n é o tamanho de x. Da mesma forma, como $L_2 \xrightarrow{\text{poli}} L_3$, qualquer instância y de L_2 pode ser convertida em tempo polinomial $q(m)$ em uma instância $g(y)$ de L_3, tal que $y \in L_2$ se e somente se $g(y) \in L_2$, onde m é o tamanho de y. Combinando essas duas construções, qualquer instância x para L_1 pode ser convertida em tempo $q(k)$ em uma instância $g(f(x))$ para L_3, tal que $x \in L_1$ se e somente se $g(f(x)) \in L_3$, onde k é o tamanho de $f(x)$. Mas $k \leq p(n)$, pois $f(x)$ é construído em $p(n)$ etapas. Assim, $q(k) \leq q(p(n))$. Como a composição de dois polinômios sempre resulta em outro polinômio, essa desigualdade implica $L_1 \xrightarrow{\text{poli}} L_3$. ■

Nesta seção, estabeleceremos que vários problemas importantes são ***NP***-completos usando esse lema. Todas as provas têm a mesma estrutura geral: dado um novo problema L, primeiramente provamos que L está em ***NP***. Em seguida reduzimos (em tempo polinomial) um problema ***NP***-completo conhecido a L, mostrando que L é ***NP***-difícil. Assim, mostramos que L está em ***NP*** e é ***NP***-difícil; portanto, L é ***NP***-completo. (Por que não fazemos a redução na direção inversa?) Essas reduções geralmente têm uma destas três formas:

- ***Restrição:*** esta forma mostra que um problema L é **NP**-difícil apresentando um problema M conhecido como ***NP***-completo como apenas um caso especial de L.
- ***Substituição local:*** esta forma reduz um problema M conhecido como ***NP***-completo a L dividindo instâncias de M e L em "unidades básicas" e depois mostrando como cada unidade básica de M pode ser convertida localmente em uma unidade básica de L.
- ***Projeto de componentes:*** esta forma reduz um problema M conhecido como ***NP***-completo a L construindo componentes para uma instância de L que exibirão funções estruturais de instâncias de M. Por exemplo, alguns componentes terão funções de "escolhas" enquanto outros terão de "avaliação".

A última das três formas acima tende a ser a mais difícil de construir. Ela é a forma usada, por exemplo, na prova do teorema de Cook-Levin (13.6).

Na Figura 13.3 ilustramos os problemas que provamos que são ***NP***-completos, junto com os problemas usados nas reduções e a técnica usada em cada redução de tempo polinomial.

No restante desta seção, estudaremos alguns problemas ***NP***-completos importantes. Trataremos a maior parte deles em pares, com cada par referindo-se a uma classe importante de problemas, incluindo problemas que envolvem fórmulas booleanas, grafos, conjuntos e números. Começaremos com dois problemas envolvendo fórmulas booleanas.

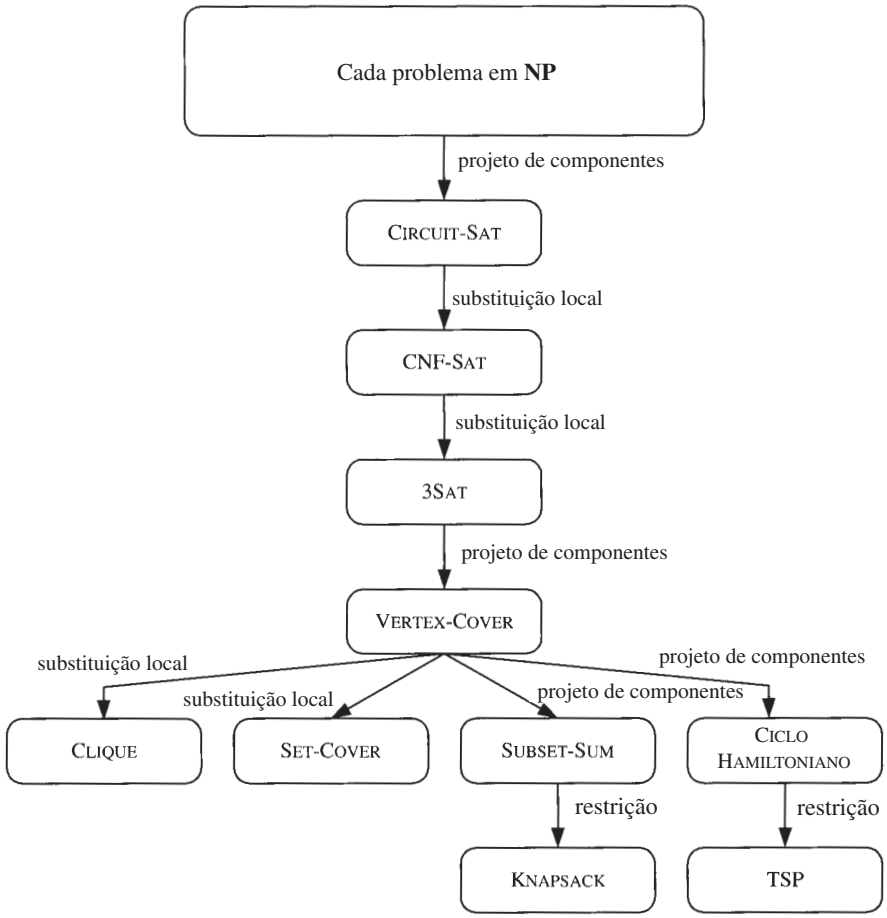


Figura 13.3 Ilustração das reduções usadas em algumas provas fundamentais de ***NP***-completude. Cada aresta dirigida denota uma redução de tempo polinomial, com o nome da aresta indicando a forma primária de redução. A primeira redução é o teorema de Cook-Levin.

13.3.1 CNF-SAT e 3SAT

As primeiras reduções que apresentaremos são para problemas envolvendo fórmulas booleanas. Uma fórmula booleana é uma expressão com parênteses que é formada a partir de variáveis booleanas que usam operações booleanas como OR (+), AND (.) e NOT (representadas como uma barra desenhada sobre a expressão sendo negada), IMPLIES (→) e IF-AND-ONLY-IF (↔). Uma fórmula booleana está em ***forma normal conjuntiva*** (CNF, em inglês*) se for constituída por uma coleção de expressões cha-

* N. de R. *Conjunctive normal form.*

madas de ***cláusulas***, que são combinadas usando AND e cada cláusula sendo formada com o uso do operador OR entre variáveis booleanas ou suas negações, ambas chamadas de ***literais***. Por exemplo, a fórmula booleana abaixo está em CNF:

$$(\overline{x_1} + x_2 + x_4 + \overline{x_7})(x_3 + \overline{x_5})(\overline{x_2} + x_4 + \overline{x_6} + x_8)(x_1 + x_3 + x_5 + \overline{x_8}).$$

Essa fórmula avalia como 1 se x_2, x_3 e x_4 forem 1, onde usamos 0 para ***falso*** e 1 para ***verdadeiro***. A CNF é chamada de forma normal porque qualquer fórmula booleana pode ser convertida para esta forma.

CNF-SAT

O problema CNF-SAT recebe uma fórmula booleana na CNF como entrada e pergunta se existe uma atribuição de valores booleanos às variáveis de maneira que a fórmula booleana tenha valor 1.

É fácil demonstrar que CNF-SAT está em ***NP***, pois, dada uma fórmula booleana S, podemos construir um algoritmo não-determinístico que "adivinha" uma atribuição de valores booleanos para as variáveis em S e avalia cada cláusula em S. Se todas as cláusulas em S tiverem valor 1, então S é satisfeita; em outro caso, S não é satisfeita.

Para demonstrar que CNF-SAT é ***NP***-difícil, reduziremos o problema CIRCUIT-SAT a CNF-SAT em tempo polinomial. Suponha que temos um circuito booleano C. Sem perda da generalidade, assumimos que cada porta AND e OR tenha duas entradas, e que cada porta NOT tenha uma entrada. Para iniciar a construção de uma fórmula S equivalente a C, criamos uma variável x_i para cada entrada em todo o circuito C. Podemos ser tentados a usar somente esse conjunto de variáveis e iniciar imediatamente a construção de S a partir de C, mas esta abordagem geralmente não tem tempo polinomial (veja o Exercício C-13.3). Em vez disso, criamos uma variável y_i para cada saída de uma porta em C. Então, criamos uma pequena fórmula booleana B_g que corresponde a cada porta g em C como segue:

- Se g for uma porta AND com entradas a e b (que poderiam ser variáveis x_i ou y_i) e saída c, então $B_g = (c \leftrightarrow a \cdot b)$.
- Se g for uma porta OR com entradas a e b e saída c, então $B_g = (c \leftrightarrow a + b)$.
- Se g for uma porta NOT com entrada a e saída b, então $B_g = (b \leftrightarrow \bar{a})$.

Desejamos criar a fórmula S tomando o AND de todos os B_g, mas uma fórmula desse tipo não estaria em CNF. Nosso método é converter cada B_g para CNF e combinar todos esses B_g transformados com operações AND para definir a fórmula booleana S em CNF.

a	b	c	$B = (c \leftrightarrow (a \cdot b))$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	1	1
0	0	1	0
0	0	0	1

Fórmula em DNF para $\overline{B} = a \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot c + \overline{a} \cdot \overline{b} \cdot c$

Fórmula em CNF para $B = (\overline{a} + \overline{b} + c) \cdot (\overline{a} + b + \overline{c}) \cdot (a + \overline{b} + \overline{c}) \cdot (a + b + \overline{c})$.

Figura 13.4 Tabela-verdade para uma fórmula booleana B com variáveis a, b e c. A fórmula equivalente a $\overline{B}$ em DNF e a fórmula equivalente a B em CNF.

Para converter uma fórmula booleana B para CNF, construímos uma tabela-verdade para B como mostrado na Figura 13.4. Então, construímos uma fórmula booleana D_i para cada linha da tabela que é avaliada em 0. Cada D_i consiste no AND das variáveis da tabela, com a variável negada se e somente se seu valor naquela linha for 0. Criamos uma fórmula booleana D fazendo o OR de todas as D_i. Uma fórmula como essa, que é o OR de fórmulas que são o AND de variáveis ou suas negações, é chamada de ***forma normal disjuntiva***, ou ***DNF***. Neste caso, temos uma fórmula D em DNF que é equivalente a $\overline{B}$, pois ela tem valor 1 se e somente se B tiver valor 0. Para converter D em uma fórmula em CNF para B, aplicamos a cada D_i as leis de De Morgan, que estabelecem que

$$\overline{(a+b)} = \overline{a} \cdot \overline{b} \quad e \quad \overline{(a \cdot b)} = \overline{a} + \overline{b}.$$

Pela Figura 13.4, podemos trocar cada B_g que seja da forma $(c \leftrightarrow a \cdot b)$ por

$$(\overline{a} + \overline{b} + c)(\overline{a} + b + \overline{c})(a + \overline{b} + \overline{c})(a + b + \overline{c}),$$

que está em CNF. De forma similar, cada B_g na forma $(b \leftrightarrow \overline{a})$ pode ser trocado pela fórmula equivalente em CNF

$$(\overline{a} + \overline{b})(a + b).$$

Deixamos a substituição da CNF por um B_g da forma $(c \leftrightarrow a + b)$ como exercício (R-13.2). Trocar cada B_g da maneira descrita resulta em uma fórmula booleana S' em CNF que corresponde exatamente a cada porta lógica e a cada entrada de C. Para construir a fórmula booleana final S, definimos $S = S' \cdot y$, onde y é a variável associada com a saída da porta que define o valor de C. Assim, C pode ser satisfeita se e somente se S puder ser satisfeita. Além disso, a construção de S a partir de C constrói uma subexpressão de tamanho constante para cada porta e cada entrada de C; portanto, essa construção tem tempo polinomial. Assim, essa redução fornece o seguinte resultado:

Teorema 13.8: CNF-SAT *é* ***NP****-completo*.

3SAT

Considere o problema 3SAT, que recebe uma fórmula booleana S em forma normal conjuntiva (CNF) com cada cláusula em S tendo exatamente três literais e que pergunta se S pode ser satisfeita. Lembre que uma fórmula booleana está em CNF se for formada pelo AND de uma coleção de cláusulas, cada uma sendo o OR de um conjunto de literais. Por exemplo, a fórmula a seguir poderia ser uma instância de 3SAT:

$$(\overline{x_1} + x_2 + \overline{x_7})(x_3 + \overline{x_5} + x_6)(\overline{x_2} + x_4 + \overline{x_6})(x_1 + x_5 + \overline{x_8}).$$

Assim, o problema 3SAT é uma versão restrita do problema CNF-SAT. (Note que não podemos usar a versão por restrição da prova de ***NP***-dificuldade, pois esta forma de prova somente funciona para reduzir uma versão restrita à sua forma mais geral.) Nesta subseção, mostramos que 3SAT é ***NP***-completo usando a forma de prova por substituição local. É interessante notar que o problema 2SAT, em que cada cláusula tem exatamente dois literais, pode ser resolvido em tempo polinomial (veja os Exercícios C-13.4 e C-13.5).

Note que 3SAT está em ***NP***, pois podemos construir um algoritmo não-determinístico de tempo polinomial que recebe uma fórmula S em CNF com três literais por cláusula e que adivinha uma atribuição de valores booleanos para S, avaliando S para verificar se o resultado é 1.

Para provar que 3SAT é ***NP***-difícil, reduzimos o problema CNF-SAT a ele em tempo polinomial. Seja C uma dada fórmula booleana em CNF. Realizamos a seguinte substituição local em cada cláusula C_i em C:

- Se $C_i = (a)$, ou seja, tiver um termo, que pode ser uma variável negada, então trocamos C_i por $S_i = (a+b+c)\cdot(a+\overline{b}+c)\cdot(a+b+\overline{c})\cdot(a+\overline{b}+\overline{c})$, onde b e c são novas variáveis não usadas anteriormente em lugar algum.
- Se $C_i = (a+b)$, ou seja, tiver dois termos, então trocamos C_i por $S_i = (a+b+c)\cdot(a+b+\overline{c})$, onde c é uma nova variável não usada anteriormente em lugar algum.
- Se $C_i = (a + b + c)$, ou seja, tiver três termos, então fazemos $S_i = C_i$.
- Se $C_i = (a_1 + a_2 + a_3 + \cdots + a_k)$, ou seja, tiver $k >$ três termos, então trocamos C_i por $(a_1 + a_2 + b_1)\cdot(\overline{b_1} + a_3 + b_2)\cdot(\overline{b_2} + a_4 + b_3)\cdots(\overline{b_{k-3}} + a_{k-1} + a_k)$, onde b_1, b_2, ..., b_{k-1} são novas variáveis não usadas anteriormente em lugar algum.

Perceba que o valor associado às variáveis introduzidas é completamente irrelevante. Não importa como elas forem avaliadas: C_i é 1 se e somente se a fórmula booleana S_i é 1. Assim, a cláusula original C é 1 se e somente se S for 1. Além disso, note que cada cláusula aumenta em tamanho em um fator constante e que as transformações feitas são simples substituições. Portanto, mostramos como reduzir uma instância de CNF-SAT a uma instância equivalente de 3SAT em tempo polinomial. Isso, junto com a observação anterior de que 3SAT pertence a ***NP***, fornece o teorema abaixo.

Teorema 13.9: 3SAT *é* ***NP****-completo.*

13.3.2 VERTEX-COVER

O Lema 13.5 lembra-nos de que VERTEX-COVER recebe um grafo G e um inteiro k e pergunta se existe uma cobertura de vértices de G contendo no máximo k vértices. Formalmente, VERTEX-COVER pergunta se existe um subconjunto C de vértices de tamanho máximo k tal que, para cada aresta (v, w), temos $v \in C$ ou $w \in C$. Mostramos, no Lema 13.5, que VERTEX-COVER está em ***NP***. O exemplo a seguir motiva esse problema.

Exemplo 13.10: *suponha que temos um grafo G representando uma rede de computadores onde os vértices representam roteadores e as arestas representam conexões. Suponha que desejamos substituir alguns dos roteadores por equipamentos novos e caros, que podem realizar operações sofisticadas de monitoração das conexões. Se desejamos determinar se k novos roteadores são suficientes para monitorar cada conexão da rede, então temos uma instância de* VERTEX-COVER.

Vamos mostrar que VERTEX-COVER é ***NP***-difícil reduzindo o problema 3SAT a ele em tempo polinomial. Essa redução é interessante em dois aspectos. Primeiramente, ela mostra um exemplo de redução de um problema lógico a um problema com grafos. Em segundo lugar, ela ilustra uma aplicação da técnica de prova por componentes.

Seja S uma instância do problema 3SAT, ou seja, uma fórmula booleana em CNF tal que cada cláusula tenha exatamente três literais. Construímos um grafo G e um inteiro k tais que G tenha uma cobertura de vértices de tamanho máximo k se e somente se S puder ser satisfeita. Começamos nossa construção acrescentando o seguinte:

- Para cada variável x_i usada na fórmula booleana S, adicionamos dois vértices a G, um rotulado como x_i e outro como $\overline{x_i}$. Também adicionamos uma aresta $(x_i, \overline{x_i})$ a G. (Nota: esses nomes são para nosso próprio benefício; depois de construir o grafo G, podemos sempre renomear os vértices com inteiros se for assim que uma instância de VERTEX-COVER deve se parecer.)

Cada aresta $(x_i, \overline{x_i})$ define o valor de x_i, pois, com esta aresta em G, uma cobertura de vértices deve incluir pelo menos um de x_i ou $\overline{x_i}$. Além disso, temos o seguinte:

- Para cada cláusula $C_i = (a + b + c)$ em S, formamos um triângulo consistindo em três vértices, $i1$, $i2$ e $i3$, e três arestas,$(i1, i2)$, $(i2, i3)$ e $(i3, i1)$.

Note que qualquer cobertura de vértices terá de incluir ao menos dois dos vértices em $\{i1, i2, i3\}$ para cada triângulo. Cada um desses triângulos é um componente que faz que uma condição seja satisfeita. Conectamos esses dois tipos de componentes adicionando, para cada cláusula $C_i = (a + b + c)$, as arestas $(i1, a)$, $(i2, b)$ e $(i3, c)$ (veja a Figura 13.5). Finalmente, avaliamos o parâmetro inteiro $k = n + 2m$, onde n é o número de variáveis em S e m é o número de cláusulas. Assim, se existir uma cobertura de vértices de tamanho máximo k, então ela deve ter tamanho exatamente igual a k. Isso completa a construção de uma instância do problema VERTEX-COVER.

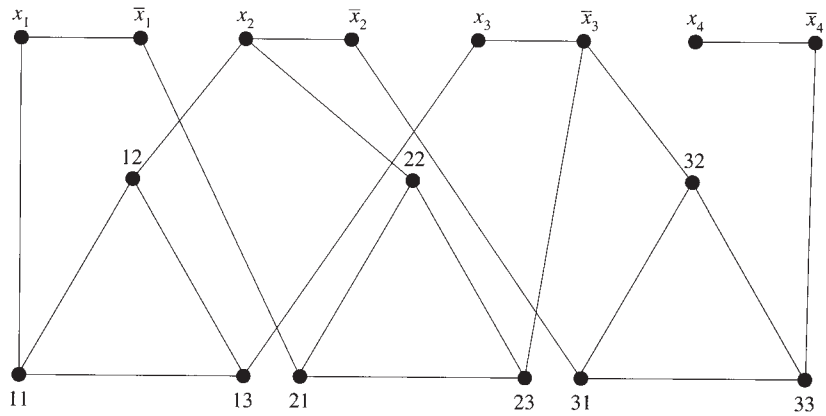


Figura 13.5 O grafo G é uma instância do problema VERTEX-COVER construído a partir da fórmula $S = (x_1 + x_2 + x_3) \cdot (\overline{x_1} + x_2 + \overline{x_3}) \cdot (\overline{x_2} + \overline{x_3} + \overline{x_4})$.

Essa construção é claramente feita em tempo polinomial, então vamos considerar se ela está correta. Suponha que exista uma atribuição de valores booleanos às variáveis de S de forma que S seja satisfeita. Com o grafo G construído a partir de S, podemos construir um subconjunto C de vértices que contenha cada literal a (em um componente que define seu valor lógico) que é associado a 1 pelo componente que garante a condição. Da mesma forma, para cada cláusula $C_i = (a + b + c)$, o componente que garante a condição atribui valor 1 a pelo menos um de a, b ou c. Seja qual for o que vale 1 (e escolhendo aleatoriamente se houver empates), inserimos os outros dois no conjunto C. O conjunto C tem o tamanho $n + 2m$. Além disso, note que cada aresta em um componente que dá valor a variáveis ou em um componente satisfatório de condições é coberta, e duas de cada três arestas que chegam a um componente satisfatório de condições também são cobertas. Adicionalmente, note que uma aresta incidente a um componente associado a uma cláusula C_i que não seja coberta por um vértice no componente deve ser coberta por um nodo em C rotulado com um literal pois o literal correspondente em C_i é 1.

Suponha o inverso, ou seja, que exista uma cobertura de vértices de C de tamanho máximo $n + 2m$. Pela construção, este conjunto deve ter tamanho exato $n + 2m$, pois ele deve conter um vértice de cada componente que dê valor a variáveis e dois vértices para cada componente que cause a satisfação de condições. Isso faz com que uma aresta seja incidente a um componente satisfatório de condições que não é coberto por um vértice no componente que dá valor a variáveis; portanto, essa aresta deve ser coberta pelo outro extremo, que é rotulado com um literal. Assim, podemos associar o literal em S associado a este nodo com 1, e cada cláusula de S é satisfeita; logo, toda a fórmula S é satisfeita. Portanto, S pode ser satisfeita se e somente se G tiver uma cobertura de vértices de tamanho máximo k. Isso permite o seguinte resultado:

Teorema 13.11: VERTEX-COVER *é* ***NP****-completo.*

Como mencionado acima, a redução ilustra a técnica de componentes. Construímos componentes para dar valor a variáveis e para a satisfação de condições em nosso grafo G a fim de garantir propriedades importantes nas cláusulas de S.

13.3.3 CLIQUE e SET-COVER

Como o problema VERTEX-COVER, existem vários problemas que envolvem a seleção de um subconjunto de objetos de um conjunto maior de forma a otimizar o tamanho que o subconjunto pode ter e ainda satisfazer alguma propriedade importante. Nesta subseção, estudamos mais dois problemas: CLIQUE e SET-COVER.

CLIQUE

Um ***clique*** em um grafo G é um subconjunto C de vértices tal que, para cada v e w em C com $v \neq w$, (v, w) é uma aresta. Ou seja, existe uma aresta entre cada par de vértices distintos em C. O problema CLIQUE recebe um grafo G e um inteiro k como entrada e pergunta se existe um clique em G de tamanho mínimo k.

Deixamos como exercício (R-13.7) mostrar que CLIQUE está em ***NP***. Para mostrar que CLIQUE é ***NP***-difícil, reduzimos o problema VERTEX-COVER a ele. Para isso, seja (G, k) uma instância do problema VERTEX-COVER. Para o problema CLIQUE, construímos o grafo complementar G^c que tem o mesmo conjunto de vértices de G, mas aresta (v, w) com $v \neq w$ se e somente se (v, w) não estiver em G. Definimos o parâmetro inteiro para CLIQUE como $n - k$, onde k é o parâmetro inteiro para VERTEX-COVER. Essa construção é feita em tempo polinomial e serve como uma redução, pois G^c tem um clique de tamanho mínimo $n - k$ se e somente se G tiver uma cobertura de vértices de tamanho máximo k (veja a Figura 13.6).

(a) (b)

Figura 13.6 Um grafo G ilustrando a prova de que CLIQUE é ***NP***-difícil. (a) mostra o grafo G com os nodos de um clique de tamanho 5 pintados em cinza. (b) mostra o grafo G^c com os nodos de um clique de tamanho 3 pintados em cinza.

Assim, temos o seguinte resultado.

Teorema 13.12: CLIQUE *é* ***NP****-completo*.

Note como a prova por substituição local acima é simples. É interessante notar que a próxima redução, que também é baseada na substituição local, é ainda mais simples.

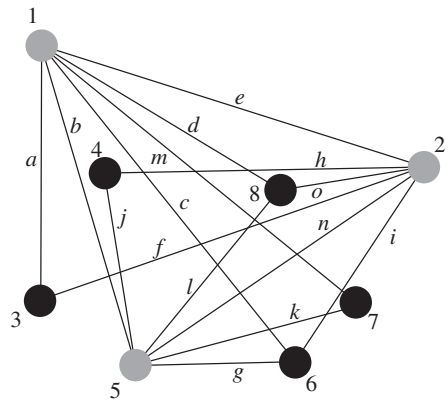
SET-COVER

O problema SET-COVER recebe uma coleção de m conjuntos $S_1, S_2, ..., S_m$ e um inteiro k como entrada e pergunta se existe uma subcoleção de k conjuntos $S_{i1}, S_{i2}, ..., S_{ik}$, tais que

$$\bigcup_{i=1}^{m} S_i = \bigcup_{j=1}^{k} S_{ij}.$$

Ou seja, a união da subcoleção de k conjuntos inclui cada elemento na união dos m conjuntos originais.

Deixamos como exercício (R-13.14) demonstrar que SET-COVER está em ***NP***. Para a redução, podemos definir uma instância de SET-COVER a partir de uma instância G e k de VERTEX-COVER: para cada vértice v de G, existe um conjunto S_v que contém as arestas de G incidentes a v. É claro que existe uma cobertura de tamanho k entre os conjuntos S_v se e somente se existir uma cobertura de vértices de tamanho k em G (veja a Figura 13.7).



(a)

$S_1 = \{a, b, c, m, d, e\}$
$S_2 = \{e, h, o, n, i\}$
$S_3 = \{a, f\}$
$S_4 = \{j, m\}$
$S_5 = \{b, j, l, n, k, g\}$
$S_6 = \{g, c, i\}$
$S_7 = \{k, m\}$
$S_8 = \{d, o, l\}$

(b)

Figura 13.7 Um grafo G ilustrando a prova de que SET-COVER é ***NP***-difícil. Os vértices são numerados de 1 a 8 e as arestas são identificadas pelas letras de a a o. (a) mostra o grafo G com os nodos de uma cobertura de vértices de tamanho 3 pintados em cinza. (b) mostra os conjuntos associados com cada vértice em G com o subscrito de cada conjunto indicando o vértice associado. Note que $S_1 \cup S_2 \cup S_5$ contém todas as arestas de G.

Assim, temos o seguinte resultado.

Teorema 13.13: SET-COVER *é* ***NP****-completo*.

Essa redução mostra como é simples converter um problema de grafos em um problema de conjuntos. Na próxima subseção, mostraremos como podemos reduzir problemas de grafos a problemas com números.

13.3.4 SUBSET-SUM e KNAPSACK

Alguns problemas difíceis envolvem apenas números. Em tais casos, devemos ter cuidado extra ao usar o tamanho da entrada em *bits*, pois alguns números podem ser mui-

to grandes. Para esclarecer o papel que o tamanho dos números pode ter, os pesquisadores dizem que um problema L é ***fortemente NP-difícil*** se L continuar ***NP***-difícil mesmo se restringirmos o valor de cada número da entrada para que ele seja limitado a um polinômio do tamanho (em *bits*) da entrada. Uma entrada x de tamanho n satisfaria essa condição, por exemplo, se cada número i em x fosse representado usando $O(\log(n))$ *bits*. Os problemas com números que examinaremos nesta seção não são fortemente ***NP***-difíceis (veja os Exercícios C-13.12 e C-13.13).

Subset-Sum

No problema Subset-Sum, temos um conjunto S de n inteiros e um inteiro k e devemos determinar se existe um subconjunto de inteiros de S que tenha soma k. Esse problema pode surgir, por exemplo, nestas circunstâncias:

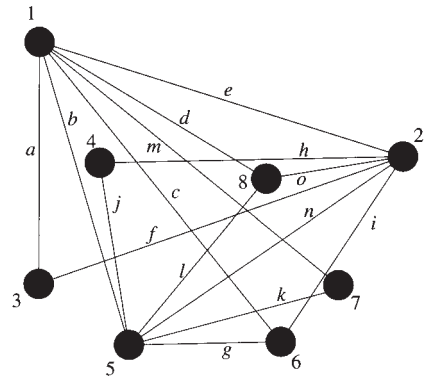
Exemplo 13.14: *suponha que temos um servidor Internet e um conjunto de pedidos de download. Para cada pedido de download, podemos determinar o tamanho do arquivo solicitado e, por isso, podemos sintetizar cada pedido como um inteiro — o tamanho do arquivo solicitado. Dado esse conjunto de inteiros, podemos estar interessados em determinar um subconjunto deles que, quando somados, tenham a largura de banda de nosso servidor durante um minuto. Infelizmente, esse problema é uma instância de Subset-Sum. Além disso, como ele é* ***NP****-completo, este problema fica mais difícil de resolver à medida que a largura de banda e a capacidade de atender pedidos aumentam.*

Subset-Sum pode parecer fácil, e, de fato, mostrar que ele está em ***NP*** é simples (veja o Exercício R-13.15). Infelizmente ele é ***NP***-completo, como mostraremos. Sejam G e k uma instância do problema Vertex-Cover. Numeramos os vértices de G de 1 a n, e as arestas de G de 1 a m e construímos a ***matriz de incidência*** H de G definida como $H[i, j] = 1$ se e somente se a aresta de número j for incidente ao vértice de número i, e $H[i, j] = 0$ em outro caso (veja a Figura 13.8).

Usamos H para definir alguns números grandes (mas ainda de tamanho polinomial) para usar como entrada para o problema Subset-Sum: para cada linha i de H que codifica todas as arestas incidentes ao vértice i, construímos o número

$$a_i = 4^{m+1} + \sum_{j=1}^{m} H[i, j]4^j.$$

Note que esse número soma uma potência de 4 diferente para cada valor 1 na i-ésima linha de H, mais uma outra grande potência de 4. A coleção de valores a_i define um "componente de incidência" para nossa redução, pois cada potência de 4 em um a_i (exceto a maior) corresponde a uma possível incidência entre o vértice i e alguma aresta.



(a)

H	1	2	3	4	5	6	7	8
a	1	0	1	0	0	0	0	0
b	1	0	0	0	1	0	0	0
c	1	0	0	0	0	1	0	0
d	1	0	0	0	0	0	0	1
e	1	1	0	0	0	0	0	0
f	0	1	1	0	0	0	0	0
g	0	0	0	0	1	1	0	0
h	0	1	0	1	0	0	0	0
i	0	1	0	0	0	1	0	0
j	0	0	0	1	1	0	0	0
k	0	0	0	0	1	0	1	0
l	0	0	0	0	1	0	0	1
m	0	1	0	1	0	0	0	0
n	0	1	0	0	1	0	0	0
o	0	1	0	0	0	0	0	1

(b)

Figura 13.8 Um grafo G ilustrando a prova de que SUBSET-SUM é ***NP***-difícil. Os vértices são numerados de 1 a 8, e as arestas recebem nomes de a a o. (a) mostra o grafo G; (b) mostra a matriz de incidência H para G. Note que existe um 1 para cada aresta em uma ou mais das colunas para os vértices 1, 2 e 5.

Além do componente anterior, também definimos um componente de "cobertura das arestas", onde, para cada aresta j, definimos o número

$$b_j = 4^j.$$

Então, calculamos a soma que desejaríamos obter de um subconjunto desses números como

$$k' = k4^{m+1} + \sum_{j=1}^{m} 2 \cdot 4^j,$$

onde k é o parâmetro para a instância de VERTEX-COVER.

Vamos considerar como essa redução, que pode ser feita em tempo polinomial, realmente funciona. Suponha que o grafo G tenha uma cobertura de vértices $C = \{i_1, i_2, \ldots, i_k\}$ de tamanho k. Então podemos construir um conjunto de valores que tenha a soma k' tomando cada a_i com um índice em C, ou seja, cada a_{i_r} para $r = 1, 2, ..., k$. Além disso, para cada aresta numerada j em G, se somente um dos pontos finais de j estiver em C, então também incluímos b_j no subconjunto. Esse conjunto de números tem soma k', pois ele inclui k valores 4^{m+1} mais dois valores 4^j (ou por dois a_{i_r} se esta aresta tiver ambas as extremidades em C ou por um a_{i_r} e um b_j se C contiver apenas uma extremidade da aresta j).

Suponha que exista um subconjunto de valores que tenha soma k'. Como k' contém k valores 4^{m+1}, ele deve incluir exatamente k valores a_i. Para cada a_i, vamos incluir o vértice i em nossa cobertura. Esse conjunto é uma cobertura, pois cada aresta j que corresponde a uma potência 4^j deve contribuir duas vezes nesta soma. Como somente um valor pode vir de um b_j, então ao menos um deve vir de um dos valores a_i escolhidos. Logo, temos o resultado a seguir.

Teorema 13.15: SUBSET-SUM *é* ***NP****-completo*.

KNAPSACK

No problema KNAPSACK, ou da mochila, ilustrado na Figura 13.9, temos um conjunto S de itens numerados de 1 a n. Cada item i tem um tamanho inteiro s_i e um valor w_i. Também temos dois parâmetros inteiros, s e w, e desejamos saber se existe um subconjunto T de S tal que

$$\sum_{i \in T} s_i \leq s, \ e \ \sum_{i \in T} w_i \geq w.$$

O problema da mochila definido acima é a versão de decisão do problema de otimização do problema da mochila 0-1, discutido na Seção 5.3.3.

Podemos motivar o problema da mochila acima com a seguinte aplicação prática.

Exemplo 13.16: *suponha que temos s artigos para vender em um leilão na Internet. Um comprador i pode fazer ofertas para diferentes lotes informando que ele está interessado em comprar s_i artigos pelo preço total de w_i unidades monetárias. Se os vários pedidos não puderem ser fracionados (ou seja, se o comprador i desejar exatamente s_i artigos), então determinar se podemos receber w unidades neste leilão dá origem ao problema da mochila. (Se os lotes puderem ser fracionados, então o problema de otimização do leilão pode ser resolvido pelo problema fracionário da mochila, que pode ser resolvido eficientemente pelos algoritmos gulosos a Seção 5.1.1.)*

O problema da mochila está em ***NP***, pois podemos construir um algoritmo não-determinístico de tempo polinomial que adivinha os itens a serem colocados no subconjunto T e verifica que eles não violam as restrições sobre s e w.

O problema da mochila também é ***NP***-difícil, pois ele contém o problema SUBSET-SUM como um caso especial. Em particular, qualquer instância de números dada para o problema SUBSET-SUM pode corresponder aos itens de uma instância do problema da mochila, com cada $w_i = s_i$ dado como um valor para o problema SUBSET-SUM e os objetivos para o tamanho s e valor w igual a k, onde k é o inteiro que desejamos somar no problema SUBSET- SUM Pela técnica de restrições, temos o resultado a seguir.

Teorema 13.17: KNAPSACK *é* ***NP****-completo.*

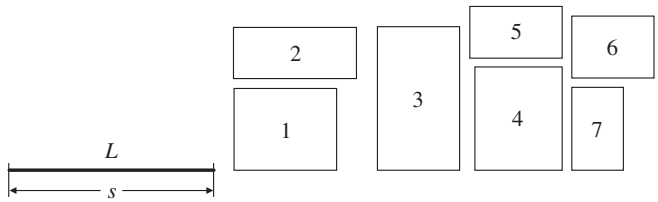


Figura 13.9 Uma percepção geométrica do problema da mochila. Dada uma linha L de comprimento s e uma coleção de n retângulos, podemos escolher um subconjunto de retângulos de forma a ter suas bordas inferiores sobre L de forma que a área total dos retângulos seja pelo menos w? Assim, a largura do retângulo i é s_i, e sua área é w_i.

13.3.5 CICLO HAMILTONIANO e TSP

Os dois últimos problemas ***NP***-completos que consideraremos envolvem a busca de certos tipos de ciclos em um grafo. Tais problemas são interessantes para otimizar o caminho de robôs e *plotters*, por exemplo.

CICLO HAMILTONIANO

Lembre, do Lema 13.3, que o CICLO HAMILTONIANO é o problema que recebe um grafo G e pergunta se existe um ciclo em G que visita cada vértice em G exatamente uma vez retornando ao vértice inicial (veja a Figura 13.10a). Também lembre, do Lema 13.3, que o CICLO HAMILTONIANO está em ***NP***. Para mostrar que este problema é ***NP***-completo, reduziremos VERTEX-COVER a ele usando uma redução baseada em componentes.

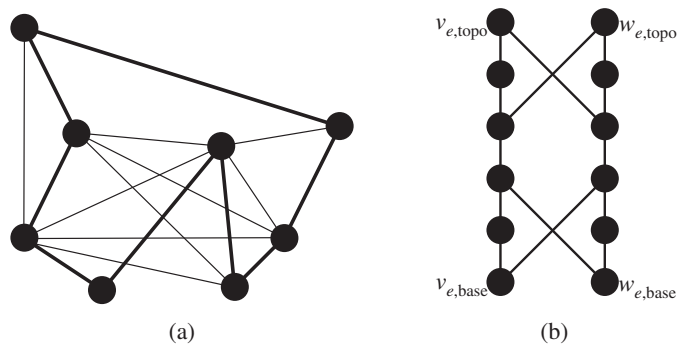


Figura 13.10 Ilustração do problema do CICLO HAMILTONIANO e sua prova de ***NP***-completude. (a) mostra um grafo com um ciclo hamiltoniano em negrito. (b) ilustra um subgrafo H_e que força a cobertura, usado para mostrar que CICLO HAMILTONIANO é ***NP***-difícil.

Sejam G e k uma instância do problema VERTEX-COVER. Construiremos um grafo H que tenha um ciclo hamiltoniano se e somente se G tiver uma cobertura de vértices de tamanho k. Começamos incluindo um conjunto de k vértices não-conexos $X = \{x_1, x_2, ..., x_k\}$ em H. Esse conjunto de vértices serve como um componente de escolha de vértices, pois eles servirão para identificar quais nodos em G devem ser incluídos em uma cobertura de vértices. Além disso, para cada aresta $e = (v, w)$ em G, criamos um subgrafo H_e em H para forçar a cobertura. Esse subgrafo H_e tem 12 vértices e 14 arestas, como mostrado na Figura 13.10b.

Seis dos vértices de H_e para $e = (v, w)$ correspondem a v, e outros seis correspondem a w. Rotulamos dois vértices em H_e correspondendo a v como $v_{e,\text{topo}}$ e $v_{e,\text{base}}$, e rotulamos dois vértices correspondendo a w como $w_{e,\text{topo}}$ e $w_{e,\text{base}}$. Esses são os únicos vértices em H_e que serão conectados a outros vértices em H fora de H_e.

Assim, um ciclo hamiltoniano pode visitar os nodos de H_e em uma de três formas possíveis, mostradas na Figura 13.11.

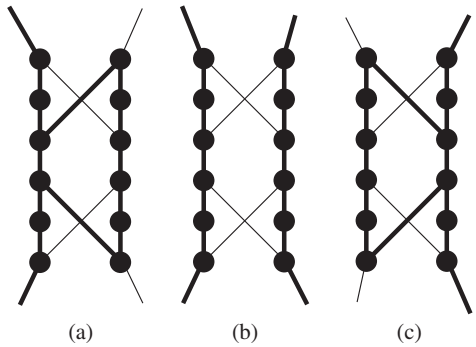


Figura 13.11 As três formas possíveis em que um ciclo hamiltoniano pode visitar as arestas em H_e.

Unimos os vértices importantes em cada subgrafo H_e a outros vértices em H de duas maneiras possíveis, uma para cada componente envolvido. Para o componente que determina a cobertura, adicionamos uma aresta de cada vértice de X a cada vértice $v_{e,\text{topo}}$ e cada vértice $v_{e,\text{base}}$. Ou seja, adicionamos $2nk$ arestas a H, onde n é o número de vértices em G.

Para o componente que força a cobertura, consideramos cada vértice v em G. Para cada v, seja $\{e_1, e_2, ..., e_{d(v)}\}$ uma lista das arestas de G que são incidentes a v. Usamos essa lista para criar arestas em H ligando $v_{e_i,\text{base}}$ em H_{ei} a $v_{e_i+1,\text{topo}}$ em H_{e_i+1}, para $i = 1, 2, ..., d-1$ (veja a Figura 13.12). Falamos dos componentes de H_{e_i} unidos desta forma como pertencendo à ***thread de cobertura*** de v. Isso completa a construção do grafo H. Note que esse processo tem tempo polinomial no tamanho de G.

Afirmamos que G tem uma cobertura de vértices de tamanho k se e somente se H tiver um ciclo hamiltoniano. Suponha que G tenha uma cobertura de vértices de tamanho k. Seja $C = \{v_{i_1}, v_{i_2}, ..., v_{i_k}\}$ uma tal cobertura. Construímos um ciclo hamiltoniano em H ligando uma série de caminhos P_j, onde cada P_j inicia em x_j e termina em x_{j+1}, para $j = 1, 2, ..., k-1$, exceto para o último caminho P_k, que inicia em x_k e termina em x_1. Formamos um caminho P_j como segue: iniciando em x_j, visitamos a *thread* de cobertura para v_{i_j} em H, retornando a x_{j+1} (ou x_1 se $j = k$). Para cada subgrafo H_e que força a cobertura na *thread* de cobertura de v_i; e que é visitado em P_j, escrevemos e sem perda da generalidade como (v_{i_j}, w). Se w não estiver em C, então visitamos H_e como na Figura 13.11a ou 13.11c (em relação a v_{i_j}). Se w estiver em C, então visitamos H_e como na Figura 13.11b. Desta forma, visitaremos cada vértice em H_e exatamente uma vez, pois C é uma cobertura de vértices para G. Assim, o ciclo construído é realmente um ciclo hamiltoniano.

Suponha, por outro lado, que H tenha um ciclo hamiltoniano. Como esse ciclo deve conter todos os vértices em X, o quebramos em k caminhos $P_1, P_2, ..., P_k$, cada um dos quais começando e terminando em um vértice de X. Além disso, pela estrutura dos subgrafos H_e e pela forma como são conectados, cada P_j deve percorrer uma parte (possivelmente toda) de uma *thread* de cobertura para um vértice v em G. Seja C o conjunto de todos estes vértices em G. Como o ciclo hamiltoniano deve incluir os vértices de cada H_e e cada um destes subgrafos deve ser percorrido de uma forma que corresponda a um (ou aos dois) extremos de e, C deve ser uma cobertura de vértices em G.

Portanto, G tem uma cobertura de vértices de tamanho k se e somente se H tiver um ciclo hamiltoniano. Assim, temos o seguinte:

Teorema 13.18: CICLO HAMILTONIANO *é* ***NP****-completo*.

TSP

No ***problema do caixeiro viajante***, ou TSP*, temos um inteiro k e um grafo G em que cada aresta e em G tem um custo inteiro $c(e)$ associado, e temos de determinar se existe um ciclo em G que visite todos os vértices (possivelmente mais de uma vez) e tenha um custo total máximo de k. Mostrar que TSP está em ***NP*** é tão simples como adivinhar a seqüência de vértices e depois verificar que ela forma um ciclo de tamanho máximo k. Mostrar que TSP é ***NP***-completo também é simples, pois ele contém o problema do CICLO HAMILTONIANO como um caso especial: para uma instância G do problema do CICLO HAMILTONIANO, podemos criar uma instância do TSP associando a cada aresta e em G um custo $c(e) = 1$ e fazendo o inteiro k igual a n, onde n é o número de vértices em G. Portanto, usando a forma de restrição de redução, temos:

Teorema 13.19: TSP *é* ***NP****-completo*.

13.4 Algoritmos aproximativos

Uma maneira de lidar com ***NP***-completude em problemas de otimização é usar um ***algoritmo aproximativo***. Um algoritmo desse tipo tipicamente é muito mais rápido do que um algoritmo que busca uma solução exata, mas não tem a garantia de encontrar a melhor solução. Nesta seção, estudaremos métodos para construir e analisar algoritmos aproximativos para problemas de otimização difíceis.

A situação genérica é que temos uma instância x de problema, que poderia ser uma codificação de um conjunto de números, um grafo, etc., como discutido anteriormente. Além disso, para o problema que estamos interessados em resolver, existe freqüentemente um grande número de soluções permitidas para x, que definiremos como o conjunto $\mathcal{F}$ de soluções permitidas.

Também temos uma função de custo c que determina um custo numérico $c(S)$ para qualquer solução $S \in \mathcal{F}$. Para um problema de otimização genérico, estamos interessados em encontrar uma solução S em $\mathcal{F}$ tal que

$$c(S) = OPT = \min \{c(T): T \in \mathcal{F}\}.$$

Ou seja, desejamos uma solução de custo mínimo. Poderíamos formular uma versão de maximização do problema de otimização, que simplesmente troca o "min" acima por "max". Para simplificar a discussão, no entanto, manteremos a idéia de que (a não ser que seja dito explicitamente) nossa meta é uma minimização.

A meta do algoritmo aproximativo é chegar o mais perto possível do valor ótimo em um tempo razoável. Como fizemos em todo este capítulo, aceitamos que um tempo razoável é no máximo um tempo polinomial.

Gostaríamos de fornecer uma garantia de quão perto a solução ótima OPT chega de um algoritmo aproximativo. Dizemos que um algoritmo fornece uma **δ-*apro-***

* N. de T. *Traveling Salesperson problem*.

ximação para um problema particular quando ele retorna uma solução permitida S (ou seja, $S \in \mathcal{F}$) tal que

$$c(S) \leq \delta OPT$$

para um problema de minimização. Para um problema de maximização, um algoritmo de δ-aproximação garantiria $OPT \leq \delta c(S)$. Em geral, temos

$$\delta \leq \max \{c(S)/OPT, OPT/c(S)\}.$$

No restante desta seção, estudaremos problemas para os quais podemos construir algoritmos de δ-aproximação para vários valores de δ. Começamos com a situação ideal no que diz respeito a aspectos da aproximação.

13.4.1 Esquemas de aproximação de tempo polinomial

Existem problemas para os quais podemos construir algoritmos de δ-aproximação que executam em tempo polinomial com $\delta = 1 + \varepsilon$, para qualquer valor fixo $\varepsilon > 0$. O tempo de execução dessa coleção de algoritmos depende de n, o tamanho da entrada, e também do valor fixo ε. Referimo-nos a essa coleção de algoritmos como um ***esquema de aproximação de tempo polinomial*** ou ***PTAS****, em inglês. Quando temos um esquema de aproximação de tempo polinomial para um problema de otimização, podemos ajustar nossa garantia de desempenho com base no tempo que desejamos gastar. Idealmente, o tempo de execução é polinomial em n e $1/\varepsilon$, caso em que temos um ***esquema de aproximação de tempo totalmente polinomial***.

Esquemas de aproximação de tempo polinomial tiram vantagem de uma propriedade que alguns problemas difíceis possuem: a de serem escaláveis. Um problema é ***escalável*** se uma instância x do problema puder ser transformada em uma instância equivalente x' (ou seja, uma com a mesma solução ótima) apenas escalando a função de custo c. Por exemplo, TSP é escalável: dada uma instância G de TSP, podemos construir uma instância equivalente G' multiplicando a distância entre cada par de vértices por um fator de escala s. O trajeto em G' será o mesmo que em G, embora o custo seja agora multiplicado por s.

Um esquema de aproximação de tempo totalmente polinomial para KNAPSACK

Para ser mais concreto, vamos fornecer um esquema de aproximação de tempo totalmente polinomial para a versão de otimização do problema KNAPSACK (Seções 5.1.1 e 13.3.4). Na versão de otimização deste problema, temos um conjunto S de itens numerados de 1 a n e um inteiro s que serve como restrição de tamanho. Cada item i em S recebe um tamanho inteiro s_i e valor w_i, e devemos determinar um subconjunto T de S tal que T maximize a soma

$$w = \sum_{i \in T} w_i \text{ e ainda satisfaça } \sum_{i \in T} s_i \leq s.$$

* N. de R. *Polynomial-time approximation scheme*.

Desejamos um PTAS que produza uma $(1 + \varepsilon)$-aproximação para qualquer constante ε. Ou seja, um algoritmo desse tipo deve encontrar um subconjunto T' satisfazendo à restrição de tamanho tal que se, definirmos $w' = \sum_{i \in T'} w_i$, então

$$OPT \leq (1 + \varepsilon)w',$$

onde OPT é a melhor soma possível em todos os subconjuntos satisfazendo à restrição de tamanho. Para provar que a desigualdade é válida, provaremos que

$$w' \geq (1 - \varepsilon/2)OPT,$$

para $0 < \varepsilon < 1$. Isso será suficiente pois, para qualquer valor fixo $0 < \varepsilon < 1$,

$$\frac{1}{1 - \varepsilon/2} < 1 + \varepsilon \ .$$

Para derivar um PTAS para KNAPSACK, tiramos vantagem do fato de ser este problema escalável. Suponha que temos um valor ε com $0 < \varepsilon < 1$. Seja $w_{\text{máx}}$ o máximo valor para um item em S. Sem perder a generalidade, assumimos que o tamanho de cada item é no máximo s (pois um item maior do que este não caberia na mochila). Assim, o item com valor $w_{\text{máx}}$ define um limite inferior para o valor ótimo, ou seja, $w_{\text{máx}} \leq OPT$. Da mesma forma, podemos definir um limite superior para o valor ótimo percebendo que a mochila pode conter no máximo todos os n itens de S, cada um dos quais com valor máximo $w_{\text{máx}}$ e, portanto, $OPT \leq nw_{\text{máx}}$. Para tirar vantagem da escalabilidade do problema da mochila, arredondamos cada valor w_i para w_i', o menor múltiplo de $M = \varepsilon w_{\text{máx}}/2n$. Vamos chamar a versão arredondada de S de S', e vamos chamar de OPT' a solução dessa versão. Note que, por simples substituição, $OPT \leq 2n^2M/\varepsilon$. Além disso, $OPT' \leq OPT$, pois arredondamos cada valor em S para formar S'. Assim, $OPT' \leq 2n^2M/\varepsilon$.

Vamos examinar nossa solução para a versão arredondada S' do problema da mochila para S. Como cada valor em S' é um múltiplo de M, qualquer valor de uma coleção de itens também será um múltiplo de M. Além disso, existem apenas $N = \lceil 2n^2/\varepsilon \rceil$ múltiplos que devem ser considerados por causa do limite superior para OPT'. Podemos usar programação dinâmica (Seção 5.3) na construção de um algoritmo eficiente para encontrar o valor ótimo para S'. Especificamente, vamos definir o parâmetro

$s[i, j]$ = o tamanho do menor conjunto de itens em $\{1, 2, ..., j\}$ com valor iM.

A percepção mais importante em um algoritmo de programação dinâmica para resolver o problema arredondado da mochila é que podemos escrever

$$s[i, j] = \min \{s[i, j-1], s_j + s[i - (w_j'/M), j-1]\}$$

para $i = 1, 2, ..., N$, e $j = 1, 2, ..., n$ (veja a Figura 13.13).

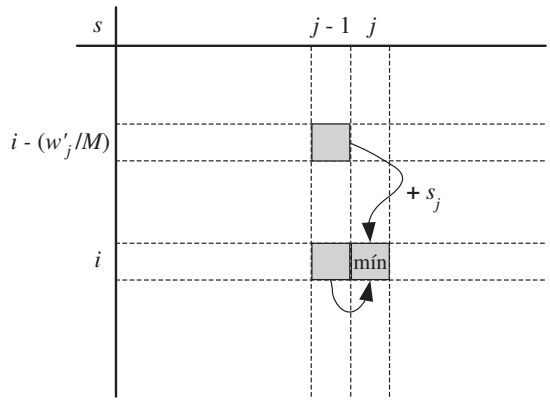


Figura 13.13 Ilustração da equação para $s[i, j]$ usada no algoritmo de programação dinâmica para a versão escalada do problema da mochila.

A equação anterior para $s[i, j]$ resulta do fato de que o item j irá contribuir ou não para a menor maneira de atingir o valor iM com os itens em $\{1, 2, ..., j\}$. Além disso, note que, para o caso base $j = 0$, quando nenhum item está na mochila, temos

$$s[i, 0] = +\infty$$

para $i = 1, 2, ..., N$. Ou seja, este tamanho não está definido. Além disso,

$$s[0, j] = 0$$

para $j = 1, 2, ..., n$ pois podemos sempre atingir o valor 0 não incluindo nenhum item na mochila. O valor ótimo é definido por

$$OPT' = \max \{iM : s[i, n] \leq s\}.$$

Esse é o valor retornado pelo nosso algoritmo PTAS.

Análise do PTAS para Knapsack

Podemos facilmente converter a descrição acima em um algoritmo de programação dinâmica que determina OPT' em tempo $O(n^3/\varepsilon)$. Um algoritmo desse tipo fornece o valor de uma solução ótima, mas podemos facilmente alterá-lo para também determinar o conjunto de itens.

Vamos considerar o benefício de uma aproximação OPT' para OPT. Lembre que reduzimos o valor w_i de cada item i em no máximo $M = \varepsilon w_{\max}/2n$. Assim,

$$OPT' \geq OPT - \varepsilon w_{\max}/2,$$

pois a solução ótima pode conter no máximo n itens. Como $OPT \geq w_{\max}$, isso implica que

$$OPT' \geq OPT - \varepsilon OPT/2 = (1 - \varepsilon/2)OPT.$$

Assim, $OPT \leq (1 + \varepsilon)OPT'$, que é o que desejávamos provar. O tempo de execução de nosso algoritmo de aproximação é $O(n^3/\varepsilon)$. A idéia de projetar um algoritmo eficiente para qualquer $\varepsilon > 0$ dá origem a um esquema de aproximação totalmente polinomial, pois o tempo de execução é polinomial em n e em $1/\varepsilon$. Esse fato nos fornece o resultado a seguir.

Teorema 13.20: *o problema de otimização* KNAPSACK *tem um esquema de aproximação totalmente polinomial que obtém uma* $(1 + \varepsilon)$*-aproximação em tempo* $O(n^3/\varepsilon)$*. onde n é o número de itens na instância de* KNAPSACK*, e* $0 < \varepsilon < 1$ *é uma constante fixa.*

13.4.2 Uma 2-aproximação para VERTEX-COVER

Não é sempre possível projetar um esquema de aproximação de tempo polinomial para um problema difícil, e menos ainda um esquema totalmente polinomial. Nesses casos, ainda desejaríamos obter uma aproximação, mas teremos de buscar um fator de aproximação que não seja arbitrariamente próximo de 1, como tivemos até o momento. Descreveremos agora como podemos obter um esquema de 2-aproximação para um problema ***NP***-completo bastante conhecido, o problema VERTEX-COVER (Seção 13.3.2). Na versão de otimização deste problema, recebemos um grafo G e devemos apresentar o menor conjunto C que seja uma cobertura de vértices para G, ou seja, que cada aresta de G seja incidente a um vértice de C.

Nosso algoritmo de aproximação é baseado no método guloso e é bastante simples. Ele envolve escolher uma aresta do grafo, colocar seus dois extremos na cobertura e deletar do grafo esta aresta e seus vértices incidentes. O processo continua até que não haja mais arestas. Damos os detalhes desta abordagem no Algoritmo 13.14.

Algoritmo VertexCoverApprox(G):
  Entrada: um grafo G.
  Saída: uma cobertura de vértices C para G.
  $C \leftarrow \emptyset$
  enquanto G tiver arestas **faça**
    escolha uma aresta $e = (v, w)$ de G
    coloque os vértices v e w em C
    para cada aresta f incidente em v ou w **faça**
      remova f de G
  retorne C

Algoritmo 13.14 Um algoritmo de 2-aproximação para VERTEX-COVER.

Deixamos os detalhes de como implementar esse algoritmo em tempo $O(n + m)$ como exercício (R-13.18). Vamos considerar agora por que esse algoritmo é uma 2-aproximação. Primeiramente, observe que cada aresta $e = (v, w)$ escolhida pelo algoritmo e usada para adicionar v e w a C deve ser considerada em qualquer cobertura de vértices, ou seja, qualquer cobertura de vértices para G deve conter v ou w (possivelmente ambos). O algoritmo de aproximação adiciona v e w a C, e quando o algoritmo termina, não existem arestas não cobertas em G, pois a cada etapa removemos as arestas cobertas pelos vértices v e w quando eles são adicionados a C. Assim, C forma uma cobertura de vértices para G. Além disso, o tamanho de C é no máximo duas vezes o tamanho de uma cobertura de vértices ótima para G, pois para cada dois vértices que adicionamos a C um deles deve pertencer à cobertura ótima. Portanto, temos o seguinte.

Teorema 13.21: *dado um grafo com n vértices e m arestas, a versão de otimização de* VERTEX-COVER *tem um algoritmo de* 2*-aproximação que custa tempo* $O(n + m)$.

13.4.3 Uma 2-aproximação para um caso especial de TSP

Na versão de otimização do problema do caixeiro viajante, ou TSP, recebemos um grafo ponderado G tal que cada aresta e em G tenha um valor inteiro $c(e)$, e devemos encontrar um ciclo de valor mínimo em G que visite todos os vértices. Nesta seção, descreveremos um algoritmo simples de 2-aproximação para um caso especial do problema de otimização de TSP.

A desigualdade triangular

Considere uma instância de TSP tal que os pesos das arestas satisfaçam a ***desigualdade triangular***, ou seja, para três arestas (u, v), (v, w) e (u, w) em G, temos

$$c((u, v)) + c((v, w)) \geq c((u, w)).$$

Também suponha que cada par de vértices em G seja ligado por uma aresta, ou seja, G é um grafo completo. Essas propriedades, que valem para qualquer noção de distância, implicam que o caminho ótimo em G visitará cada vértice exatamente uma vez; portanto, vamos considerar apenas ciclos hamiltonianos como soluções possíveis para o TSP.

O algoritmo de aproximação

Nosso algoritmo de aproximação tira vantagem das propriedades descritas de G para propor um algoritmo de três fases. Na primeira fase, construímos uma árvore de cobertura mínima M para G (Seção 7.3). Na segunda fase, construímos um caminho de Euler E para os nodos da árvore M, ou seja, um caminhamento de M que comece e termine no mesmo vértice e percorra cada aresta de M exatamente uma vez em cada direção (Seção 2.3.3). No terceiro passo, construímos um ciclo T a partir de E caminhando pelas arestas de E, mas cada vez que tivermos duas arestas (u, v) e (v, w) em E, tais que v já tenha sido visitado, substituímos essas duas arestas pela aresta (u, w) e continuamos. Essencialmente estamos construindo T como um caminhamento de preordenação de M. Esse algoritmo de três fases claramente tem tempo polinomial (veja a Figura 13.15).

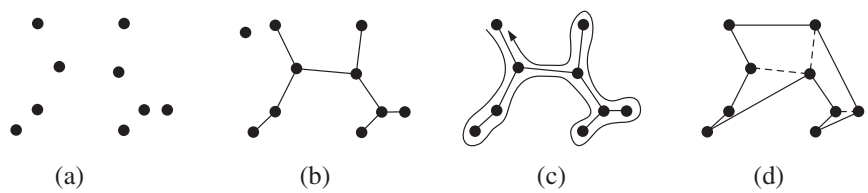


Figura 13.15 Exemplo do algoritmo de aproximação para TSP em um grafo que satisfaz a desigualdade triangular: (a) um conjunto S de posições no plano, com a distância euclidiana definindo os custos das arestas (não mostradas); (b) a árvore de cobertura mínima M para S; (c) um caminho euleriano E para M; (d) um caminho aproximado T para TSP.

Análise do algoritmo de aproximação para TSP

A análise do motivo pelo qual este algoritmo atinge um fator de aproximação 2 também é simples. Vamos estender nossa notação de forma que $c(H)$ denote o peso total

das arestas de um subgrafo H de G. Seja T', o caminho ótimo para o grafo G. Se deletarmos qualquer aresta de T', teremos um caminho que também é uma árvore de cobertura mínima. Assim,

$$c(M) \leq c(T').$$

Também podemos relacionar facilmente o custo de E ao custo de M, como

$$c(E) = 2c(M),$$

pois o caminho de Euler E visita cada aresta de M somente uma vez em cada direção. Finalmente, note que pela desigualdade triangular cada vez que substituímos duas arestas (u, v) e (v, w) por (u, w) durante a construção de T, não aumentamos o custo do caminho, ou seja,

$$c(T) \leq c(E).$$

Portanto, temos

$$c(T) \leq 2c(T')$$

(Veja a Figura 13.16.)

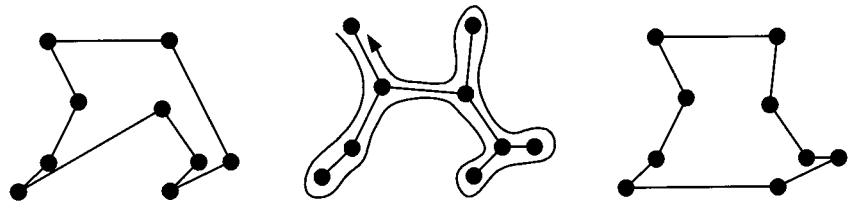


Figura 13.16 Ilustração da prova de que o algoritmo baseado em MST é uma 2-aproximação para o problema de otimização de TSP.

Resumimos a discussão a seguir.

Teorema 13.22: *se um grafo ponderado G for completo e tiver arestas com pesos que satisfaçam a desigualdade triangular, então existe um algoritmo de* 2*-aproximação para o problema de otimização de* TSP *em G, e esse algoritmo tem tempo polinomial.*

Esse teorema depende fortemente do fato de que os custos das arestas no grafo G satisfazem a desigualdade triangular. De fato, sem essa hipótese, nenhum algoritmo de aproximação de fator constante e tempo polinomial pode existir para a versão de otimização de TSP, a não ser que $\boldsymbol{P} = \boldsymbol{NP}$ (veja o Exercício C-13.14).

13.4.4 Uma aproximação logarítmica para SET-COVER

Existem casos em que obter até mesmo uma aproximação de fator constante em tempo polinomial é difícil. Nesta seção, estudaremos um dos mais conhecidos desses problemas, o SET- COVER (Seção 13.3.3). Na versão de otimização deste problema, temos uma coleção de conjuntos $S_1, S_2, ..., S_m$, cuja união é um universo U de tamanho n,

e desejamos encontrar o menor inteiro k tal que exista uma subcoleção de k conjuntos $S_{i_1}, S_{i_2}, ..., S_{i_k}$ com

$$U = \bigcup_{i=1}^{m} S_i = \bigcup_{j=1}^{k} S_{ij}.$$

Embora seja difícil encontrar um algoritmo de aproximação de fator constante que execute em tempo polinomial para esse problema, podemos propor um algoritmo eficiente que tenha um fator de aproximação $O(\log(n))$. Como em vários outros algoritmos de aproximação para problemas difíceis, esse algoritmo é baseado no método guloso (Seção 5.1).

Uma abordagem gulosa

O algoritmo escolhe conjuntos S_{i_j} um de cada vez, sempre escolhendo o conjunto que tenha o máximo de elementos não cobertos. Quando todos os elementos de U estiverem cobertos, terminamos. Fornecemos uma descrição simples em pseudocódigo no Algoritmo 13.17.

Algoritmo SetCoverApprox(S):
  Entrada: uma coleção S de conjuntos $S_1, S_2, ..., S_m$ cuja união é U.
  Saída: uma cobertura de conjuntos C para S.
  $C \leftarrow \emptyset$ {a cobertura que estamos construindo }
  $E \leftarrow \emptyset$ {o conjunto de elementos já cobertos de U }
  enquanto $E \neq U$ **faça**
    escolha um conjunto S_i de S que tenha o maior número de elementos não cobertos e adicione S_i a C
    $E \leftarrow E \cup S_i$
  retorne C

Algoritmo 13.17 Um algoritmo de aproximação para SET-COVER.

Esse algoritmo tem tempo polinomial (veja o Exercício R-13.19).

Análise do tempo do algoritmo SET-COVER

Para analisar o fator de aproximação do algoritmo guloso para SET-COVER, usaremos um argumento de amortização baseado em um esquema de cobrança (Seção 1.5): cada vez que nosso algoritmo de aproximação selecionar um conjunto S_j, cobraremos os elementos de S_j pela seleção.

Especificamente, considere o momento em que um conjunto S_j é adicionado a C, e seja k o número de elementos não cobertos em C'. Devemos pagar um total de \$ 1 para adicionar esse conjunto a C, e cobraremos isto dos elementos não cobertos dividindo \$ 1 entre os i elementos não cobertos de S_j:

$$c(i) = 1/k$$

Assim, o tamanho total da nossa cobertura é igual ao total de cobranças feitas pelo algoritmo, ou seja,

$$|C| = \sum_{i \in U} c(i).$$

Para provar um limite para essa aproximação, vamos considerar as cobranças feitas aos elementos em cada subconjunto S_j que pertença a uma cobertura ótima C'. Assim, suponha que S_j pertença a C'. Vamos escrever $S_j = \{x_1, x_2, ..., x_{n_j}\}$ de forma que os elementos de S_j sejam listados na ordem em que são cobertos por nosso algoritmo (empates são decididos arbitrariamente). Agora considere a iteração em que x_1 é coberto. Neste momento, S_j ainda não foi selecionado; portanto, qualquer conjunto selecionado deve ter ao menos n_j elementos não cobertos. Assim, cobra-se de x_1 no máximo o valor $1/n_j$. Vamos considerar então o momento em que o algoritmo cobra um elemento x_l de S_j. No pior caso, ainda não teremos escolhido S_j (pode ser que nosso algoritmo nunca o escolha). Qualquer que seja o conjunto escolhido nesta iteração, ele tem, no pior caso, pelo menos $n_j - l + 1$ elementos não cobertos; portanto, x_ℓ é cobrado em no máximo $1/(n_j - l + 1)$. Assim, o total cobrado de todos os elementos de S_j é no máximo

$$\sum_{l=1}^{n_j} \frac{1}{n_l - l + 1} = \sum_{l=1}^{n_j} \frac{1}{l} ,$$

que é o familiar número harmônico H_{nj}. É sabido (para exemplo, veja o Apêndice) que H_{nj} é $O(\log n_j)$. Seja $c(S_j)$ o total cobrado de todos os elementos de S_j que pertençam à cobertura ótima C'. Nosso esquema de cobrança implica que $c(S_j)$ seja $O(\log n_j)$, e somando para todos os conjuntos de C', temos

$$\begin{aligned} \sum_{S_j \in C'} c(S_j) &\leq \sum_{S_j \in C'} b \ \log n_j \\ &\leq b|C'| \ \log n, \end{aligned}$$

para alguma constante $b \geq 1$. Mas como C' é uma cobertura de conjuntos,

$$\sum_{i \in U} c(i) \leq \sum_{S_j \in C'} c(S_j).$$

Portanto,

$$|C| \leq b|C'| \ \log n.$$

Isso fornece o resultado a seguir.

Teorema 13.23: *a versão de otimização do problema* SET-COVER *tem um algoritmo de* $O(\log(n))$*-aproximação de tempo polinomial para encontrar uma cobertura de uma coleção de conjuntos cuja união é um universo de tamanho n.*

13.5 *Backtracking* e *branch-and-bound*

Nas seções anteriores, mostramos muitos problemas sendo ***NP***-completos. Assim, a não ser que ***P = NP***, o que a maior parte dos cientistas da computação acredita não ser verdade, é impossível resolver qualquer um desses problemas em tempo polinomial. Mesmo assim, muitos deles surgem em aplicações na vida real, soluções devem ser encontradas mesmo se isso custar muito tempo. Portanto, nesta seção vamos mostrar técnicas de lidar com ***NP***-completude que têm se mostrado promissoras na prática. Essas técnicas permitem desenvolver algoritmos que encontrem soluções para problemas difíceis – com freqüência, em uma quantidade de tempo razoável. Nesta seção, estudaremos os métodos ***backtracking*** e ***branch-and-bound***.

13.5.1 *Backtracking*

O projeto com *backtracking* é uma forma de construir um algoritmo para um problema difícil L. Esse algoritmo procura sistematicamente em um grande (possivelmente exponencial) conjunto de possibilidades. A estratégia de procura é geralmente otimizada para evitar simetrias em instâncias para L e para percorrer o espaço de busca de forma que seja encontrada uma solução "fácil" para L, se uma tal solução existir.

A técnica de ***backtracking*** tira vantagem da estrutura inerente a muitos problemas ***NP***-completos. Lembre que a aceitação de uma instância x de um problema em ***NP*** pode ser feita em tempo polinomial, dado um certificado de tamanho polinomial. Freqüentemente esse certificado consiste em uma série de escolhas tais como os valores para variáveis booleanas, um subconjunto de vértices de um grafo ou um conjunto de objetos para colocar em uma mochila. De forma similar, a verificação de um certificado freqüentemente envolve um teste simples para verificar se o certificado fornece uma configuração bem-sucedida para x, tal como satisfazer uma fórmula, cobrir todas as arestas de um grafo ou estar de acordo com algum critério de desempenho. Nesses casos, podemos usar um algoritmo de ***backtracking*** dado no Algoritmo 13.18 para procurar sistematicamente pela solução do problema, se esta solução existir.

O algoritmo de *backtracking* percorre possíveis "caminhos de procura" para localizar soluções ou "becos sem saída". A configuração no final de um caminho consiste em um par (x, y), onde x é o subproblema que ainda deve ser resolvido, e y é o conjunto de escolhas feitas para chegar a esse subproblema a partir do problema original. Inicialmente fornecemos ao algoritmo de *backtracking* o par $(x, \emptyset)$, onde x é o problema original. Se o algoritmo descobrir em algum momento que uma configuração (x, y) não conduz a uma solução válida, não importando quantas escolhas adicionais sejam feitas, então ele evita todas as procuras a partir dessa configuração e retorna à outra. Na verdade, esta abordagem dá aos algoritmos de *backtracking* seu nome*.

Algoritmo Backtrack(x):
  Entrada: uma instância x de um problema difícil.
  Saída: uma solução para x ou "sem solução" se nenhuma existir.
  $F \leftarrow \{(x, \emptyset)\}$ {F é o conjunto de "fronteira" de configurações de subproblemas}
  enquanto $F \neq \emptyset$ **faça**
    retire de F a configuração (x, y) mais promissora
    expanda (x, y) fazendo um pequeno conjunto de escolhas adicionais
    sejam $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$ o conjunto de novas configurações
    para cada nova configuração (x_i, y_i) **faça**
      verifique a consistência de (x_i, y_i)
      se a verificação retornar "solução encontrada" **então**
        retorne a solução derivada de (x_i, y_i)
      se a verificação retornar "sem saída" **então**
        descarte a configuração (x_i, y_i) { Backtrack }
      senão
        $F \leftarrow F \cup \{(x_i, y_i)\}$ {(x_i, y_i) inicia uma procura mais promissora}
  retorne "sem solução"

Algoritmo 13.18 O padrão para o algoritmo de *backtracking*.

* N. de R. *Backtrack* significa recuar, voltar atrás.

Preenchendo os detalhes

Para transformar a estratégia de *backtracking* em um algoritmo realmente operacional, precisamos apenas preencher os detalhes a seguir:

1. Definir uma forma de escolher a configuração mais promissora dentro do conjunto de fronteira F.
2. Especificar a forma de expansão de uma configuração (x, y) em configurações para subproblemas. Este processo de expansão deve ser capaz de gerar todas as configurações possíveis iniciando pela configuração inicial $(x, 0)$.
3. Descrever como realizar uma verificação de consistência para uma configuração (x, y) que retorne "solução encontrada" ou "sem saída".

Se F for uma pilha, então podemos fazer uma pesquisa em profundidade no espaço de configuração. De fato, neste caso podemos usar a recursão para implementar F automaticamente como uma pilha. Se F for uma fila, teremos uma pesquisa em largura no espaço de configuração. Podemos também imaginar outras estruturas de dados para implementar F; porém, enquanto tivermos essa noção intuitiva de como selecionar a configuração mais "promissora" de F a cada iteração teremos um algoritmo de *backtracking*.

Para tornar essa abordagem mais concreta, trabalharemos com uma aplicação desta técnica para o problema CNF-SAT.

Um algoritmo com *backtracking* para CNF-SAT

Lembre que no problema CNF-SAT temos uma fórmula booleana S em forma normal conjuntiva (CNF) e desejamos saber se S pode ser satisfeita. Para propor um algoritmo com *backtracking* para CNF-SAT, faremos sistematicamente tentativas de atribuição para as variáveis em S e verificaremos se essas atribuições fazem S ser avaliada para 1 ou 0 ou se fornecem uma nova fórmula S' para a qual continuaremos fazendo atribuições de valores às variáveis. Assim, uma configuração em nosso algoritmo consiste em um par (S', y), onde S' é uma fórmula booleana em CNF e y é uma atribuição de valores para variáveis que não estão em S', de forma que, fazendo estas atribuições em S, obtemos S'.

Para formular o algoritmo de *backtracking*, precisamos fornecer os detalhes de cada um dos três componentes de um algoritmo de *backtracking*. Dada a fronteira F de configurações, fazemos a escolha mais promissora como a fórmula S' com a menor cláusula. Essa é a mais restrita de todas as fórmulas em F; portanto, esperamos chegar a um beco sem saída mais depressa, se isto tiver de acontecer.

Vamos considerar então como gerar subproblemas a partir de uma subfórmula S'. Fazemos isto localizando a menor cláusula C em S' e escolhendo uma variável x_i que apareça em C. Então criamos dois subproblemas que sejam associados com as atribuições $x_i = 1$ e $x_i = 0$, respectivamente.

Finalmente devemos definir como processar S' para efetuar a verificação de consistência para uma atribuição da variável x_i em S'. Começamos reduzindo quaisquer cláusulas contendo x_i, usando o valor 0 ou 1 atribuído a x_i. Se a redução resultar em uma nova cláusula com um único literal x_j ou $\overline{x_j}$, também realizamos a atribuição adequada a x_j, para fazer com que a fórmula seja satisfeita. Em seguida processamos a fórmula resultante para propagar o valor atribuído a x_j e, se isso resultar em uma nova cláusula com apenas um literal, repetimos o processo até que não existam fórmulas

com apenas um literal. Se a qualquer momento descobrirmos uma contradição (ou seja, cláusulas x_i e $\overline{x_i}$, ou uma cláusula vazia), então retornamos "sem saída". Se reduzirmos a subfórmula S' até a constante 1, então retornamos "solução encontrada" e as atribuições de variáveis que nos levaram a este ponto. Em qualquer outro caso, derivamos uma nova subfórmula S'' tal que cada cláusula tenha pelo menos dois literais e as atribuições de variáveis que nos levaram de S até S''. Chamamos essa operação de ***redução*** por propagar a atribuição de um valor a x_i em S'.

Ajustar todas essas peças no padrão para um algoritmo de *backtracking* resulta em um algoritmo que resolve o problema CNF-SAT o mais rápido quanto possível. Em geral, o tempo de execução de pior caso para este algoritmo é exponencial, mas o *backtracking* freqüentemente pode acelerar o processo. De fato, se cada cláusula na fórmula tiver no máximo dois literais, então este algoritmo tem tempo polinomial. (veja o Exercício C-13.4).

Exemplo em Java: uma solução com *backtracking* para SUBSET-SUM

Para ser ainda mais concreto, vamos considerar uma solução com *backtracking* em Java para resolver o problema ***NP***-difícil chamado de SUBSET-SUM. Lembre na Seção 13.3.4 que no problema SUBSET-SUM temos um conjunto S de n inteiros e um inteiro k, e devemos determinar se existe um subconjunto de inteiros em S cuja soma seja k.

Para decidir mais facilmente se uma configuração é um beco sem saída, vamos assumir que os inteiros em S sejam dados em ordem não-decrescente como $S = \{a_0, a_1, ..., a_{n-1}\}$. Feita essa escolha, definimos os três componentes principais do algoritmo de *backtracking* como segue:

1. Para escolher a configuração mais promissora, fazemos uma escolha comum a muitos algoritmos com *backtracking*: usar recursão para fazer uma pesquisa em profundidade no espaço de configurações. Assim, nosso método controla automaticamente as configurações ainda não exploradas.
2. Para expandir uma configuração (x, y) em configurações para subproblemas, vamos apenas seguir a ordem dada dos números em S. Assim, se uma configuração já tiver considerado o subconjunto $S_i = \{a_0,...,a_i\}$, então apenas temos de gerar as duas configurações possíveis, usando a_{i+1} ou não.
3. Para realizar uma verificação de consistência que retorne "beco sem saída", "solução encontrada" ou apenas "continue", usamos outra vez o fato de que os inteiros em S estão ordenados. Suponha que já consideramos os inteiros no subconjunto $S_i = \{a_0,...,a_i\}$ e estamos agora considerando a_{i+1}. Existe um teste simples de consistência que podemos usar neste momento: seja k_i a soma de elementos em S_i que já escolhemos para nossa seleção, e seja

$$r_{i+1} = \sum_{j=i+1}^{n-1} a_j$$

a soma dos inteiros restantes. A primeira parte do nosso teste é confirmar que

$$k \leq k_i + a_{i+1}$$

Se essa condição falhar, então estamos superestimando k. Além disso, como S é ordenado, qualquer a_j com $j > i + 1$ também irá superestimar k, então estamos em um beco sem saída, neste caso. Para a segunda parte do teste, vamos confirmar que

$$k \leq k_i + r_{i+1}.$$

Se essa condição falhar, então estamos subestimando k e não poderemos atingir k com os inteiros restantes em S. Portanto, estamos em outro beco sem saída. Se passarmos em ambos os testes, então consideramos a_{i+1} e prosseguimos.

Fornecemos o código Java para esse método no Trecho de Código 13.19.

```
/**
 * Método para achar um subconjunto de um arranjo de inteiros com soma k
   assumindo que:
 * – o arranjo está ordenado de forma não-decrescente
 * – já consideramos os elementos até o índice i
 * – sum guarda a soma dos elementos que escolhemos
 * – reamin guarda a soma dos elementos que não escolhemos
 * – A função retorna "true" e imprime o subconjunto se ele for encontrado.
 * – Deve ser chamada como findSubset (a,k,-1,0,t), onde t é o total.
 */
public static boolean findSubset(int[ ] a, int k, int i, int sum, int remain) {
 /* Testa condições para expandir a configuração
   if (i+1 >= a.length) return false;          //verificação de que ainda restam
                                               valores
   if (sum + remain < k) return false;         // estamos subestimando k
   int next = a[i+1];        // o proximo candidato
   if (sum + next > k) return false;           // estamos superestimando k
   if (sum + next = = k) {              // achamos uma solução
     System. out. print(k + "="+ next);        // imprime a solução
     return true;
     }
   if (findSubset(a, k, i+1, sum+next, remain-next)) {
     System.out.print("+" +  next);             // a solução inclui a[i+1]
   return true;
   }
else    // backtracking – a solução não inclui a[i+1]
   return findSubset(a, k, i+1, sum, remain);
}
```

Trecho de Código 13.19 Implementação Java de um algoritmo de *backtracking* para o problema SUBSET-SUM.

Note que definimos recursivamente o método de procura findSubset e usamos a recursão para fazer o *backtracking* e para imprimir uma solução quando ela for encontrada. Note que cada chamada recursiva consome tempo $O(1)$, pois o arranjo é passado por referência, e assim o tempo de execução de pior caso deste método é $O(2^n)$, onde n é o número de inteiros considerados. Esperamos, é claro, que para instâncias típicas do problema, o sistema de *backtracking* evite esse pior caso, seja achando rapidamente uma solução ou detectando becos sem saída e evitando muitas configurações impossíveis.

Existe uma melhora pequena que podemos fazer no método findSubset se confiarmos que o método é sempre chamado corretamente: podemos omitir o teste de segurança para valores que ainda restam a ser considerados, pois se não houver mais intei-

ros a considerar, então remain = 0. Assim, sum + remain < k, pois teríamos encerrado este caminho em busca de uma solução antes se tivéssemos superestimado k ou acertado seu valor exatamente. Em teoria também poderíamos omitir o teste de que ainda restam inteiros, mas é melhor mantê-lo, por razões de segurança, caso o método seja chamado incorretamente.

13.5.2 *Branch-and-bound*

O algoritmo de *backtracking* é eficiente para problemas de decisão, mas não foi planejado para problemas de otimização, em que, além de termos de satisfazer alguma condição de validade para um certificado y associado a uma instância x, também desejamos que uma dada função $f(x)$ seja maximizada ou minimizada (sem perda da generalidade, vamos assumir que a função deve ser minimizada). Mesmo assim, podemos estender o algoritmo de *backtracking* para trabalhar com um problema de otimização e, ao fazer isso, obteremos o padrão de algoritmos chamado de ***branch-and-bound***.

O padrão *branch-and-bound** dado no Algoritmo 13.20 tem todos os elementos de *backtracking*, mas ele não termina simplesmente ao achar a primeira solução. Em vez disso, continuamos até a melhor solução ser encontrada. Além disso, o algoritmo tem um mecanismo de pontuação para escolher sempre a configuração mais promissora a ser explorada em cada iteração. Por causa dessa abordagem, *branch-and-bound* é chamado às vezes de estratégia de busca ***best-first***.**

Algoritmo Branch-and-Bound(x):
  Entrada: uma instância x para um problema difícil de otimização (minimização)
  Saída: uma solução ótima para x, ou "sem solução" se uma solução não existir,
  $F \leftarrow \{(x, \emptyset)\}$ { conjunto de fronteira das configurações }
  $b \leftarrow (+\infty, \emptyset)$ { custo e configuração da melhor solução conhecida }
  enquanto $F \neq \emptyset$ **faça**
    retire de F a configuração (x, y) mais promissora
    expanda (x, y) obtendo um conjunto $(x_1, y_1), ..., (x_k, y_k)$ de configurações adicionais
    para cada nova configuração (x_i, y_i) **faça**
      verifique a consistência de (x_i, y_i)
      se a verificação retornar "solução encontrada" **então**
        se o custo c da solução para (x_i, y_i) for melhor do que b **então**
          $b \leftarrow (c, (x_i, y_i))$
        senão
          descarte a configuração (x_i, y_i)
      se a verificação retornar "sem saída" **então**
        descarte a configuração (x_i, y_i) { backtrack }
      senão
        se $lb(x_i, y_i)$ é menor do que o custo de b **então**
          $F \leftarrow F \cup \{(x_i, y_i)\}$ $\{(x_i, y_i)$ inicia um novo caminho promissor $\}$
        senão
          descarte a configuração (x_i, y_i) {o valor é alto demais e já não interessa}
    retorne b

* N. de R. Em inglês, "ramificar e limitar".
** N. de R. Em inglês, "o primeiro melhor".

Algoritmo 13.20 Esquema para um algoritmo *branch-and-bound*. Esse algoritmo assume a existência de uma função $lb(x_i, y_i)$ que retorna um limite inferior para o custo de qualquer solução derivada da configuração (x_i, y_i).

Para fornecer um critério de otimização e poder escolher a configuração mais "promissora", adicionamos mais uma hipótese às três hipóteses usadas para um algoritmo de *backtracking*:

- para qualquer configuração (x, y), assumimos que temos uma função $lb(x_i, y_i)$ que retorna um limite inferior para o custo de qualquer solução derivada desta configuração.

A única exigência sobre $lb(x_i, y_i)$ é que ela seja menor ou igual ao custo de qualquer solução derivada. Como deve ser claro a partir do algoritmo de *branch-and-bound*, quanto mais preciso for esse limite inferior, mais eficiente será o algoritmo.

Um algoritmo de *branch-and-bound* para TSP

Para tornar a abordagem por *branch-and-bound* mais concreta, vamos considerar como ela pode ser aplicada para resolver a versão de otimização do problema do caixeiro viajante (TSP). Na versão de otimização deste problema, recebemos um grafo G com uma função de custo $c(e)$ definida para cada aresta e de G e desejamos encontrar o ciclo de menor custo total que visite todos os vértices de G e retorne ao vértice inicial.

Podemos propor um algoritmo para TSP se calcularmos o caminho de menor custo para cada aresta $e = (v, w)$, com o caminho começando em v, terminando em w e visitando todos os outros vértices. Para achar esse caminho, aplicamos a técnica *branch-and-bound*: geramos o caminho de v a w em $G - \{e\}$ aumentando o caminho atual em um vértice a cada passagem no laço do algoritmo *branch-and-bound*.

- Depois que construímos um caminho parcial P iniciando em v, consideramos apenas formas de aumentar o caminho com vértices fora de P.
- Podemos classificar um caminho parcial P como um "beco sem saída" se os vértices fora de P não forem conexos em $G - \{e\}$.
- Para definir a função lb que avalia um limite inferior, podemos usar o custo total das arestas em P mais $c(e)$. Isso certamente será um limite inferior para qualquer caminho que seja construído com e e com P.

Adicionalmente, depois que executamos o algoritmo para uma aresta e em G, podemos usar o melhor caminho encontrado até o momento para todas as arestas testadas, em vez de reiniciar a melhor solução b em $+\infty$. O tempo de execução do algoritmo resultante ainda será exponencial no pior caso, mas na prática ele evitará um total considerável de avaliações desnecessárias. O problema de TSP é de considerável interesse e tem muitas aplicações, então uma solução como essa poderia ser útil, na prática, se o número de vértices no grafo de entrada não fosse muito grande. Além disso, existem várias outras heurísticas que podem ser adicionadas à procura de um caminho ótimo para o TSP, mas elas estão além do escopo deste livro.

Exemplo em Java: uma solução *branch-and-bound* para KNAPSACK

Para ser ainda mais concreto, vamos descrever uma implementação em Java de uma solução *branch-and-bound* para o problema da mochila (KNAPSACK). Na versão de otimização deste problema, temos um conjunto S de itens numerados de 0 a $n - 1$. Cada item i tem um tamanho inteiro s_i e valor w_i. Também temos um parâmetro inteiro *size* e devemos determinar um subconjunto T de S tal que $\sum_{i \in T} s_i \leq size$, e o valor total dos itens em T, $valor = \sum_{i \in T} w_i$, seja maximizado.

Vamos começar tomando emprestadas algumas idéias dos algoritmos gulosos para resolver o problema fracionário da mochila (Seção 5.1.1): vamos assumir que os itens em S são colocados em ordem não-decrescente, pelo valor de w_i/s_i. Eles serão processados nesta ordem, pois assim estaremos considerando os itens na ordem de benefício decrescente, começando com o elemento de maior benefício, ou seja, o elemento com maior valor para seu tamanho. Nossa configuração é definida pelo subconjunto S_i dos primeiros i elementos de S baseados nesta ordem. Assim, os índices dos itens de S_i estão entre 0 e $i - 1$, e podemos definir o conjunto S_0 como uma configuração vazia de índice –1.

Iniciamos colocando a configuração S_0 em uma fila de prioridade P e, a cada iteração do algoritmo *branch-and-bound*, escolhemos a configuração mais promissora c em P. Se i é o índice do último item considerado em c, então expandimos c em duas novas configurações – uma que inclui o item $i + 1$ e uma que não o inclui. Note que cada configuração satisfatória à restrição de tamanho é uma configuração válida para o problema da mochila. Assim, se qualquer uma das duas configurações for válida e melhor do que a melhor solução achada até o momento, atualizamos nossa melhor opção corrente e continuamos o processo.

Para escolher configurações que sejam mais promissoras, precisamos ter uma forma de avaliá-las por seu valor potencial. Como no problema da mochila estamos interessados em maximizar o valor em vez de minimizar um custo, avaliamos as configurações calculando um limite superior para seu valor potencial: dada uma configuração c que considera os itens de índices 0 a i, calculamos um limite superior para c iniciando com o valor total w_c de c e verificando quanto valor a mais podemos colocar em c se aumentarmos c com uma solução para o problema fracionário da mochila que seja retirada dos itens restantes em S. Lembre que os itens em S estão ordenados pelo valor de w_i/s_i, e seja k o maior índice tal que $\sum_{j=i+j}^{k} s_j \leq size - s_c$, onde s_c é o tamanho de todos os itens já na configuração c. Os elementos de índices $i + 1$ a k são os melhores itens que ainda cabem na mochila. Para calcular o limite superior para c, consideramos a adição de todos esses elementos a c mais tudo o que for possível do item $k + 1$ (se este existir). Nosso limite superior para c será então definido como segue:

$$upper(c) = w_c + \sum_{j=i+1}^{k} w_j + \left(size - s_c - \sum_{j=i+1}^{k} s_j \right) \frac{w_k + 1}{s_k + 1} \ .$$

Se $k = n - 1$, então assumimos que $w_{k+1}/s_{k+1} = 0$.

Fornecemos trechos de código para a solução Java baseada nessa abordagem *branch-and-bound* nos Trechos de Código 13.21, 13.22 e 13.23.

```
/   **
    * Método para achar uma solução ótima do problema KNAPSACK, dados:
    * - s, um arranjo indexado contendo os tamanhos
    * - w, um arranjo indexado contendo o benefício dos elementos
```

```
* - índices de s e w são ordenados pelos valores de w[i]/s[i]
* - size é a restrição de tamanho dada
* - retorna um objeto Configuração contendo a melhor solução.*/
  public static Configuration solve(int[ ] s, int[ ] w, long size) {
  /* Cria fila de prioridade para escolher a melhor configuração atual */
  PriorityQueue p = DoublePriorityQueue();
  /* Cria configuração raiz */
  Configuration root = new Configuration (s,w,size);
  double upper = root.getUpperBound(); // limite superior para a raiz
  Configuration curBest = root;  // melhor solução corrente
  p.insertItem(new Double(–upper), root);  // adiciona configuração da raiz a p
  /* gera novas configurações até que todas as configurações viáveis sejam
  achadas*/
  while (!p.isEmpty()) {
    double curBound = –((Double)p.minKey()).doubleValue();  //desejamos o
    máximo
    Configuration curConfig = (Configuration) p.removeMin();
    if (curConfig.getIndex() >= s.length–1) continue;  // nada a expandir
    /* Expande esta configuração para incluir o próximo item */
    Configuration child = curConfig.expandWithNext();
    /* Testa se novo filho tem o melhor valor até agora */
    if ((child.getWorth() > curBest.getWorth()) && (child.getSize() <= size))
      curBest = child;
    /* Testa se vale a pena expandir o próximo item */
    double newBound = child.getUpperBound();
    if (newBound > curBest.getWorth())
      p.insertItem( new Double(–newBound), child);
    /  * Expande a configuração corrente para excluir o próximo item */
    child = curConfig.expandWithoutNext();
    /  * Testa se vale a pena expandir o próximo item */
    newBound = child.getUpperBound();
    if (newBound > curBest.getWorth())
      p.insertItem(new Double(–newBound), child);
  }
  return curBest;
}
```

Trecho de Código 13.21 Método principal para a solução com *branch-and-bound* para o problema da mochila. A classe DoublePriorityQueue não é mostrada, mas implementa uma fila de prioridade especializada para conter objetos com chaves Double. Note que os valores de chave usados na fila de prioridade são tornados negativos, pois desejamos maximizar o valor e não minimizar o custo. Também note que, quando expandimos uma configuração sem adicionar mais um elemento, não verificamos se esta é uma solução melhor, pois ela vale o mesmo que sua configuração-pai.

```
class Configuration {
    protected int index;    // índice do último elemento considerado
    protected boolean in;       // verdadeiro se e somente se o último elemento es-
                                   tiver na solução atual
    protected long worth; // valor total de todos os elementos desta solução
    protected long size;    // tamanho total de todos os elementos desta solução
    protected Configuration parent;// configuração gerando esta solução
    protected static int[ ] s;
    protected static int[ ] w;
    protected static long bagSize;
    /** A configuração inicial – chamada apenas para a configuração raiz */
    Configuration (int[ ] sizes, int[ ] worths, long sizeConstraint) {
      /* Cria referências estáticas às restrições válidas para todas as configurações */
      s = sizes;
      w = worths;
      bagSize = sizeConstraint;
      /* Coloca valores na configuração raiz */
      index = –1;
      in = false;
      worth = 0L;
      size = 0L;
      parent = null;
    }
    /** Construtor default */
    Configuration()          { /* Assume valores iniciais default */}
    /** Expande esta configuração para uma que contém o próximo item */
    public Configuration expandWithNext() {
      Configuration c = new Configuration();
      c.index = index + 1;
      c.in = true;
      c.worth = worth + w[c.index];
      c.size = size + s[c.index];
      c.parent = this;
      return c;
}
/** Expande esta configuração para uma que não contém o próximo item */
public Configuration expandWithoutNext() {
    Configuration c = new Configuration();
    c.index = index + 1;
    c.in = false;
    c.worth = worth;
    c.size = size;
    c.parent = this;
    return c;
}
```

Trecho de Código 13.22 Classe Configuration e seus métodos de construção e expansão de configurações. Eles são usados na versão branch-and-bound do problema da mochila. Esta classe tem continuação no Trecho de Código 13.23.

```
/** Determina o índice desta configuração */
public long getIndex() {
  return index;
}
/** Determina o tamanho desta configuração */
public long getSize() {
  return size;
}
/** Determina o valor desta configuração */
public long getWorth() {
  return worth;
}
/** Determina o limite superior para o benefício desta configuração */
public double getUpperBound() {
  int g; // índice para a solução gulosa
  double bound = worth;          // inicia do benefício atual
  long curSize=0L;
  long sizeConstraint = bagSize – size;
  /* Adiciona (de forma gulosa) itens até que o limite de tamanho seja
  ultrapassado */
  for (g=index+1; (curSize <= sizeConstraint) && (g < s.length); g++) {
    curSize += s[g];
    bound += (double) w[g];
    }
  if (g < s.length) {
    bound –= w[g];       // recua até um benefício que caiba na configuração
    /* Adiciona fração do item adicional */
    bound += (double) (bagSize – size) *w[g]/s[g];
  }
  return bound;
}
/** Imprime solução a partir desta configuração */
public void printSolution() {
  Configuration c = this;          // Inicia com a configuração dada
  System.out.printin("(Size,Worth) = " + c.size + "," + c.worth);
  System.out.print(" index-size-worth list = [");
  for (; c.parent != null; c = c.parent)       // vai até a raiz
    if (c.in)     {// imprime índice, tamanho e valor do próximo item incluído
      System.out.print("(" + c.index);
      System.out.print("," + s[c.index]);
      System.out.print("," + w[c.index] + ")");
    }
  System.out.printin("]");
}
```

Trecho de Código 13.23 Métodos de suporte para a classe Configuration do Trecho de Código 13.22 usados na solução *branch-and-bound* do problema da mochila. O método para calcular limites superiores é especialmente importante.

13.6 Exercícios

Reforço

R-13.1 O professor Entrenós mostrou que um problema de decisão L pode ser reduzido em tempo polinomial a um problema ***NP***-completo M. Além disso, após 80 páginas de matemática complicada, ele mostrou que L pode ser resolvido em tempo polinomial. Ele acabou de provar que ***P*** = ***NP***? Por que, ou por que não?

R-13.2 Use uma tabela-verdade para converter a fórmula booleana $B = (a \leftrightarrow (b + c))$ em uma fórmula equivalente em CNF. Mostre a tabela-verdade e a fórmula em DNF intermediária para $\overline{B}$.

R-13.3 Demonstre que o problema SAT, que recebe uma fórmula booleana S como entrada e pergunta se S pode ser satisfeita, é ***NP***-completo.

R-13.4 Considere o problema DNF-SAT, que recebe uma fórmula booleana em forma normal disjuntiva (DNF) como entrada e pergunta se S pode ser satisfeita. Descreva um algoritmo determinístico de tempo polinomial para DNF-SAT.

R-13.5 Considere o problema DNF-DISSAT, que recebe uma fórmula booleana em forma normal disjuntiva (DNF) como entrada e pergunta se S não pode ser satisfeita, ou seja, se existe uma atribuição de valores booleanos às variáveis de S de forma que S seja avaliada em 0. Mostre que DNF-DISSAT é ***NP***-completo.

R-13.6 Converta a fórmula booleana $B = (x_1 \leftrightarrow x_2) \cdot (\overline{x_3} + x_4 x_5) \cdot (\overline{x_1 x_2} + x_3 \overline{x_4})$ para CNF.

R- 13.7 Demonstre que o problema CLIQUE é ***NP***.

R- 13.8 Dada a fórmula booleana em CNF $B = (x_1) \cdot (\overline{x_2} + x_3 + x_5 + \overline{x_6}) \cdot (x_1 + x_4) \cdot (x_3 + \overline{x_5})$, demonstre a redução de B em uma entrada equivalente para o problema 3SAT.

R- 13.9 Dado $B = (x_1 + \overline{x_2} + x_3) \cdot (x_4 + x_5 + \overline{x_6}) \cdot (x_1 + \overline{x_4} + \overline{x_5}) \cdot (x_3 + x_4 + x_6)$, desenhe a instância de VERTEX-COVER que é construída pela reduçao de 3SAT para a fórmula booleana B.

R-13.10 Desenhe um exemplo de um grafo com 10 vértices e 15 arestas que tem uma cobertura de vértices de tamanho 2.

R-13.11 Desenhe um exemplo de um grafo com 10 vértices e 15 arestas que tem uma clique de tamanho 6.

R-13.12 O professor Entrenós acaba de projetar um algoritmo que recebe um grafo G com n vértices e determina em tempo $O(n^k)$ se G contém ou não uma clique de tamanho k. O professor Entrenós merece o Prêmio Turing por ter acabado de mostrar que ***P*** = ***NP***? Por que ou por que não?

R-13.13 Existe um subconjunto dos números {23, 59, 17, 47, 14, 40, 22, 8} que some 100? E 130? Mostre como você obteve sua resposta.

R-13.14 Demonstre que SET-COVER está em ***NP***.

R-13.15 Demonstre que SUBSET-SUM está em ***NP***.

R-13.16 Desenhe um exemplo de um grafo com 10 vértices e 20 arestas que tenha um ciclo hamiltoniano. Também desenhe um grafo que com 10 vértices e 20 arestas que não tenha um ciclo hamiltoniano.

R-13.17 A ***distância de Manhattan*** entre dois pontos (a, b) e (c, d) é definida como $|a - c| + |b - d|$. Usando a distância de Manhattan para definir o custo entre cada par de pontos, encontre um caminho ótimo para o caixeiro viajante para o conjunto de pontos a seguir: $\{(1, 1), (2, 8), (1, 5), (3, -4), (5, 6), (-2, -6)\}$.

R-13.18 Descreva em detalhes como implementar o Algoritmo 13.14 em tempo $O(m + n)$ para um grafo de n vértices com m arestas. Você pode usar a técnica tradicional de contagem de operações para a medida do tempo de execução neste caso.

R-13.19 Descreva em detalhes uma implementação eficiente do Algoritmo 13.17 e analise seu tempo de execução.

R-13.20 Forneça um exemplo de um grafo G com pelo menos 10 vértices de forma que o algoritmo guloso de 2-aproximação para VERTEX-COVER garantidamente produza uma solução não-ótima.

R-13.21 Forneça um grafo completo e ponderado G tal que os pesos de suas arestas satisfaçam à desigualdade triangular, mas que o algoritmo de aproximação baseado na MST não encontre uma solução ótima.

R-13.22 Forneça uma descrição em pseudocódigo do algoritmo de *backtracking* para CNF-SAT.

R- 13.23 Forneça em pseudocódigo uma descrição recursiva do algoritmo de *backtracking*, assumindo que a estratégia de procura visite as configurações por pesquisa em profundidade.

R-13.24 Forneça uma descrição em pseudocódigo do algoritmo baseado em *branch-and-bound* para TSP.

R-13.25 O programa baseado em *branch-and-bound* da Seção 13.5.2 para resolver o problema da mochila usa um *flag* booleano para determinar quando um item é incluído em uma solução ou não. Mostre que esse *flag* é redundante, e que se for removido, existe uma maneira (sem usar campos adicionais) de dizer se um item está incluído na solução ou não.

Criatividade

C-13.1 Demonstre que podemos simular deterministicamente em tempo polinomial qualquer algoritmo não-determinístico A que tenha tempo polinomial e faça no máximo $O(\log(n))$ chamadas ao método choose, onde n é o tamanho da entrada de A.

C-13.2 Demonstre que toda linguagem L em P pode ser reduzida em tempo polinomial à linguagem $M = \{5\}$, ou seja, à linguagem que simplesmente pergunta se a codificação binária da entrada é igual a 5.

C-13.3 Demonstre como construir um circuito booleano C tal que, se criamos variáveis somente para as entradas de C e depois tentarmos construir uma fórmula booleana equivalente a C, então criaremos uma fórmula exponencialmente maior do que uma codificação de C.

Dica: use recursão para repetir subexpressões de forma que seu tamanho dobre cada vez que são usadas.

C-13.4 Demonstre que o algoritmo de *backtracking* dado na Seção 13.5.1 para o problema CNF-SAT tem tempo polinomial se cada cláusula na fórmula booleana tiver no máximo dois literais, ou seja, se ele resolver 2SAT em tempo polinomial.

C-13.5 Considere a versão 2SAT do problema CNF-SAT, em que cada cláusula da fórmula booleana S tem dois literais. Note que qualquer cláusula da forma $(a + b)$ pode ser vista como duas implicações: $(\overline{a} \rightarrow b)$ e $(\overline{b} \rightarrow a)$. Considere um grafo G a partir de S, onde cada vértice de G esteja associado a uma variável x em S ou sua negação $\overline{x}$. Suponha que exista uma aresta dirigida em G de $\overline{a}$ para b para cada cláusula equivalente a $(\overline{a} \rightarrow b)$. Demonstre que S não pode ser satisfeita se e somente se existir uma variável x tal que exista um caminho em G de x a $\overline{x}$ e um caminho de $\overline{x}$ a x. Obtenha desta regra um algoritmo de tempo polinomial para resolver este caso especial do problema CNF-SAT. Qual o tempo de execução de seu algoritmo?

C-13.6 Suponha que um oráculo deu a você um computador mágico C que recebe qualquer fórmula booleana B em CNF e responde em uma etapa se B pode ser satisfeita ou não. Demonstre como usar C para construir uma atribuição de valores booleanos que satisfaçam qualquer fórmula booleana B que possa ser satisfeita. Quantas chamadas você precisa fazer a C para fazer isto, no pior caso?

C-13.7 Defina SUBGRAPH-ISOMORPHISM como o problema em que temos grafos G e H e determinamos se H é um subgrafo de G, ou seja, se existe um mapeamento de cada vértice v em H para um vértice $f(v)$ em G tal que, se (v, w) é uma aresta em H, então $(f(v), f(w))$ é uma aresta em G. Mostre que SUBGRAPH-ISOMORPHISM é ***NP***-completo.

C-13.8 Defina INDEPENDENT-SET como o problema que recebe um grafo G e um inteiro k e pergunta se G contém um conjunto de vértices independentes de tamanho k. Ou seja, se G contém um conjunto I de vértices de tamanho k tal que para quaisquer v e w em I não existe uma aresta (v, w) em G. Mostre que INDEPENDENT-SET é ***NP***-completo.

C-13.9 Defina HYPER-COMMUNITY como o problema que recebe uma coleção de n páginas da Web e um inteiro k e determina se existem k páginas que contêm links de uma para a outra. Mostre que HYPER-COMMUNITY é ***NP***-completo.

C-13.10 Defina PARTITION como o problema que recebe um conjunto $S = \{s_1, s_2, ..., s_n\}$ de números e pergunta se existe um subconjunto T de S tal que

$$\sum_{s_i \in T} s_i = \sum_{s_i \in S-T} s_i.$$

Ou seja, ele pergunta se existe uma partição dos números em dois grupos que tenham a mesma soma. Mostre que PARTITION é ***NP***-completo.

C-13.11 Mostre que o problema CICLO-HAMILTONIANO em grafos dirigidos é ***NP***-completo.

C-13.12 Mostre que o problema SUBSET-SUM pode ser resolvido em tempo polinomial se a entrada for dada em codificação unária. Ou seja, mostre que SUBSET-SUN não é fortemente ***NP***-difícil. Qual o tempo de execução de seu algoritmo?

C-13.13 Mostre que o problema da mochila pode ser resolvido em tempo polinomial se a entrada for dada em codificação unária. Ou seja, mostre que KNAPSACK não é fortemente ***NP***-difícil. Qual o tempo de execução de seu algoritmo?

C-13.14 Considere a versão generalizada de otimização do problema TSP, em que o grafo não precisa satisfazer a desigualdade triangular. Mostre que para qualquer valor fixo $\delta \geq 1$, não existe um algoritmo de δ-aproximação de tempo polinomial para o problema generalizado, a não ser que $\boldsymbol{P} = \boldsymbol{NP}$.

Dica: reduza o CICLO-HAMILTONIANO a este problema definido uma função de custo para um grafo completo H para o grafo de entrada G com n vértices de forma que as arestas de H que também estão em G tenham custo 1 mas arestas de H que não estão em G tenham custo $1 + \delta\, n$.

C-13.15 Considere o caso especial de TSP em que os vértices correspondam a pontos no plano e o custo definido para a aresta entre os pontos p e q seja a distância euclidiana entre os pontos. Mostre que um caminho ótimo não tem arestas que se cruzam.

C-13.16 Obtenha um algoritmo eficiente com *backtracking* para o problema do CICLO-HAMILTONIANO.

C-13.17 Obtenha um algoritmo eficiente com *backtracking* para o problema de decisão da mochila.

C-13.18 Obtenha um algoritmo com *branch-and-bound* para o problema de otimização da mochila.

C-13.19 Obtenha uma nova função limitante *lb* para um algoritmo *branch-and-bound* para resolver o problema de otimização TSP. Sua função deve ser sempre maior ou igual à função *lb* usada na Seção 13.5.2, mas ainda assim deve ser uma função limitante válida. Descreva um exemplo em que sua função seja estritamente maior do que a função usada na Seção 13.5.2.

Projetos

P-13.1 Projete e implemente um algoritmo de *backtracking* para o problema CNF-SAT. Compare o tempo de execução de seu algoritmo em várias instâncias de 2SAT e 3SAT.

P-13.2 Projete e implemente um algoritmo *branch-and-bound* para o problema de TSP. Use pelo menos duas definições diferentes para a função *lb* que calculem um limite inferior, e teste a efetividade de cada uma.

P-13.3 Trabalhando em grupo, projete e implemente um algoritmo *branch-and-bound* para o problema TSP, assim como um algoritmo de aproximação de tempo polinomial para TSP. Teste a eficiência das duas implementações para encontrar caminhos para conjuntos de pontos do plano com a distância euclidiana entre eles definindo o custo entre os pares.

P-13.4 Implemente um algoritmo de *backtracking* para CICLO-HAMILTONIANO. Para vários valores de n, teste sua eficiência para encontrar ciclos hamiltonianos quando o número de arestas é $2n$, $\lceil n \log n \rceil$, $10\,n$ e $20\lceil n^{1,5} \rceil$.

P-13.5 Faça uma comparação empírica do uso de programação dinâmica e *backtracking* para o problema SUBSET-SUM.

P-13.6 Faça uma comparação empírica do uso de programação dinâmica e *branch-and-bound* para o problema da mochila.

Notas

Modelos de computação são discutidos nos livros-texto de Lewis e Papadimitriou [133], Savage [177] e Sipser [187].

A prova do teorema de Cook-Levin (13.6) dada neste capítulo é uma adaptação de uma prova dada em Cormen, Leiserson e Rivest [55]. O teorema original de Cook [53] mostrou que CNF-SAT é ***NP***-completo, e o teorema original de Levin [131] era para um problema de colocação de ladrilhos. Chamamos o Teorema 13.6 de teorema de "Cook-Levin'' em homenagem a estes dois trabalhos iniciais, pois suas provas seguem as mesmas linhas da prova do Teorema 13.6. Karp [113] demonstrou que vários outros problemas são ***NP***-completos e, subseqüentemente, centenas de outros problemas foram demonstrados como ***NP***-completos. Garey e Johnson [76] fornecem uma interessante discussão da ***NP***-completude e um catálogo de muitos problemas ***NP***-completos e ***NP***-difíceis importantes.

As reduções dadas neste capítulo que usam substituição local e restrição são bem conhecidas na literatura de computação: por exemplo, veja Garey e Johnson [76] ou Aho, Hopcroft e Ullman [7]. A prova de projeto de componentes de que VERTEX-COVER é ***NP***-completo é uma adaptação de uma prova de Garey e Johnson [76], o mesmo se aplicando à prova de projeto de componentes de que CICLO-HAMILTONIANO é ***NP***-completo. Esta é uma combinação de duas reduções de Karp [113]. A prova de projeto de componentes de que SUBSET-SUM é ***NP***-completo é uma adaptação de uma prova dada em Cormen, Leiserson e Rivest [55].

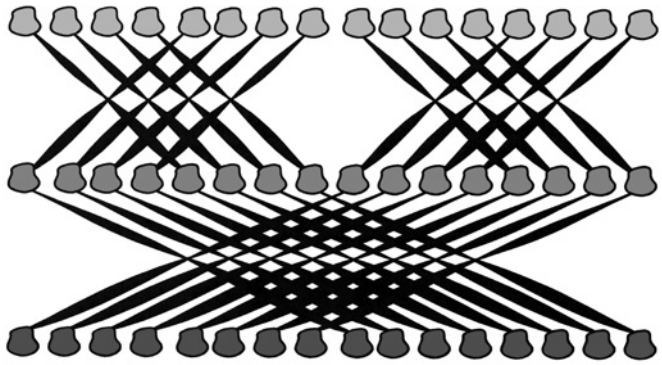
A discussão sobre *backtracking* e *branch-and-bound* é modelada segundo as discussões de Lewis e Papadimitriou [133] e Brassard e Bratley [38], em que o *backtracking* é usado para problemas de decisão e *branch-and-bound*, para problemas de otimização. Nossa discussão também é influenciada por Neapolitan e Naimipour [159], que interpretam *backtracking* como uma busca heurística que usa uma pesquisa em profundidade de um espaço de configuração, e *branch-and-bound* como uma busca heurística que usa pesquisa em largura ou *first-best* com uma função de limite inferior para fazer a poda do número de opções. A técnica de *backtracking* data do trabalho inicial de Golomb e Baumert [80].

Discussões genéricas sobre algoritmos de aproximação podem ser encontradas em vários outros livros, incluindo os de Hochbaum [97] e Papadimitriou e Steiglitz [165], bem como o capítulo de Klein e Young [116]. O PTAS para o problema da mochila segue um resultado de Ibarra e Kim [106], como apresentado por Klein e Young [116]. Papadimitriou e Steiglitz atribuem a 2-aproximação para VERTEX-COVER a Gavril e Yannakakis. O algoritmo de 2-aproximação para o caso especial de TSP é de Rosenkrantz, Stearns e Lewis [174]. A $O(\log(n))$-aproximação para SET-COVER e sua prova são conseqüências do trabalho de Chvátal [46], Johnson [109] e Lovász [136].

Capítulo

Frameworks algorítmicos

14



14.1 Algoritmos de memória externa 635

14.1.1 Gerenciamento de memória hierárquica 636

14.1.2 Árvores (a, b) e árvores-B 639

14.1.3 Ordenação em memória externa 643

14.2 Algoritmos paralelos 646

14.2.1 Modelos paralelos de computação 646

14.2.2 Divisão e conquista paralela simples 648

14.2.3 Subconjuntos seqüenciais e o teorema de Brent 649

14.2.4 Recursão dupla 650

14.2.5 Junção e ordenação paralelas 653

14.2.6 Determinação do diâmetro de um polígono convexo 654

14.3 Algoritmos *on-line* 655

14.3.1 Algoritmos de *cache* 656

14.3.2 Estratégias de leilão 662

14.3.3 Árvores de pesquisa competitivas 664

14.4 Exercícios 668

Na maior parte deste livro, estivemos presos a um *framework* computacional. Esse *framework* é formalmente conhecido como máquina de acesso direto (ou RAM) e consiste em um único processador conectado a uma única memória potencialmente ilimitada. Além disso, nos concentramos basicamente no estudo de algoritmos que aceitam entradas únicas e processam essas entradas para gerar uma saída. Esse *framework* foi útil, modelando a maioria dos problemas que os projetistas de algoritmos encontram. Mas ele tem suas limitações na medida em que existem diferentes contextos computacionais em que esse *framework* não se aplica. Estudaremos três desses *frameworks* neste capítulo.

O primeiro *framework* que relacionamos é uma extensão do componente de memória do modelo RAM. Neste *framework*, também chamado de modelo da ***memória estendida***, tentamos modelar de forma mais realística a hierarquia de memória presente nos computadores modernos. Em particular, tentamos modelar o fato de que a memória está dividida em uma memória interna rápida, que tem tempo de acesso curto e baixa capacidade, e uma memória externa lenta, com tempo de acesso longo, porém com grande capacidade. Projetar algoritmos eficientes com essa hierarquia de memória em mente requer algumas modificações nas técnicas usadas quando assumimos que toda memória tinha o mesmo tempo de acesso. Neste capítulo, examinamos algumas alterações específicas que podem ser feitas para pesquisa e ordenação a fim de tornar esses algoritmos mais eficientes em memória externa.

Outra variação importante no *framework* RAM tradicional que iremos considerar são os algoritmos paralelos. Ou seja, consideremos algoritmos que possam usar vários processadores em conjunto para resolver um problema. No projeto de algoritmos paralelos, desejamos soluções que sejam melhores em relação aos algoritmos seqüenciais conhecidos por um fator que seja o mais próximo possível de um fator linear em relação ao número de processadores. Estudaremos vários algoritmos paralelos importantes, incluindo algoritmos para aritmética paralela, pesquisa e ordenação, procurando maneiras de explorar a extensão paralela do modelo RAM que permita múltiplos processadores.

O último *framework* que examinaremos neste capítulo desafia o ponto de vista de que um algoritmo seja apenas uma função que mapeia entradas em saídas. No *framework* de algoritmos *on-line*, consideramos um algoritmo como um servidor que deve processar uma seqüência de requisições de clientes ao longo do tempo. A resposta de uma requisição pode ser totalmente processada antes que possamos examinar a próxima. Este modelo é motivado pela forma como os computadores são normalmente usados na Internet para processar requisições computacionais de usuários remotos. O desafio de projetar algoritmos para este modelo é que a escolha que fazemos para uma requisição pode aumentar o tempo necessário para processar uma requisição futura. Para analisar a efetividade das escolhas de um algoritmo *on-line*, então, devemos comparar seu comportamento com aquele de um algoritmo *off-line* que conhece a seqüência de requisições dos clientes previamente. Tal análise é conhecida como análise de competição e forma uma analogia natural com o pior caso para complexidade de tempo, que temos usado ao longo deste livro, em algoritmos que mapeiam as entradas em saídas.

14.1 Algoritmos de memória externa

Existem várias aplicações de computador que têm de lidar com grandes quantidades de dados. Exemplos incluem a análise de conjuntos de dados científicos, o processamento de transações financeiras e organização e manutenção de bancos de dados (tais como catálogos telefônicos). Na verdade, a quantidade de dados com que se lida é, em geral, muito grande para caber na memória interna do computador.

A hierarquia de memória

De maneira a acomodar grandes conjuntos de dados, os computadores têm uma ***hierarquia*** de diferentes tipos de memórias, que variam em termos de tamanho e distância da CPU. Próximos à CPU, estão os registradores internos usados pela própria CPU. O acesso a essas localizações é muito rápido, mas elas são relativamente poucas. No segundo nível da hierarquia, está a memória ***cache***. Esta memória é consideravelmente maior do que o conjunto de registradores da CPU, mas o tempo de acesso é maior (e podem existir múltiplas *caches* com tempo de acesso progressivo). Em um terceiro nível da hierarquia, encontramos a ***memória interna***, que também é conhecida como ***memória principal***, ***memória núcleo*** ou memória de acesso direto (RAM). A memória interna é consideravelmente maior do que a memória *cache*, mas também tem um tempo de acesso maior. Finalmente, no nível mais alto da hierarquia, está a ***memória externa***, que normalmente consiste em discos, CDs ou fitas. Esta memória é muito grande, mas também muito lenta. Assim, a hierarquia de memória de um computador pode ser vista como formada por quatro níveis, cada um maior e mais lento do que o anterior (veja a Figura 14.1).

Na maioria das aplicações, entretanto, apenas dois níveis realmente importam – aquele que pode conter todos os dados de um problema e o nível imediatamente abaixo. Movimentar os dados para dentro e para fora da memória de mais alto nível que pode conter todos os itens é tipicamente um gargalo computacional, neste caso.

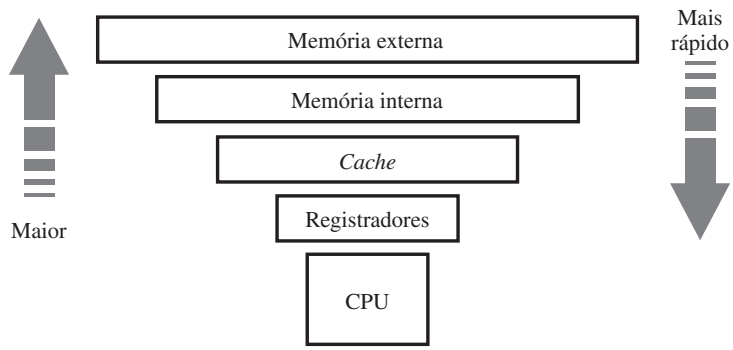


Figura 14.1 Hierarquia de memória.

Caches e discos

Os níveis específicos que importam dependem basicamente do tamanho do problema que pretendemos resolver. Para um problema que pode ser inteiramente acomodado na memória principal, os dois níveis importantes são a memória *cache* e a memória inter-

na. O tempo de acesso da memória interna pode ser de 10 a cem vezes maior do que o da memória *cache*. É desejável, por isso, executar a maioria dos acessos à memória em memória *cache*. Já para um problema que não cabe na memória principal, por outro lado, os dois níveis importantes são a memória interna e a memória externa. Neste caso, as diferenças no tempo de acesso são ainda mais dramáticas, pois os dispositivos de memória externa usuais são tipicamente de 100 mil a 1milhão de vezes mais lentos do que a memória interna.

Para colocar este último caso em evidência, imagine que há um estudante em Baltimore que deseja enviar uma mensagem pedindo dinheiro a seus pais em Chicago. Se o estudante enviar uma mensagem eletrônica, ela pode chegar a seu computador em casa em cinco segundos. Pense neste modo de comunicação como sendo o acesso da CPU a uma memória interna. O modo de comunicação correspondendo ao acesso à memória externa que é 500 mil vezes mais lento; seria o mesmo que o estudante caminhar até Chicago e entregar a mensagem pessoalmente, o que levaria um mês se ele pudesse caminhar 20 milhas por dia. Assim, devemos fazer o mínimo de acessos possíveis à memória externa.

Nesta seção, discutiremos estratégias de gerenciamento de memória hierárquica e apresentaremos métodos de ordenação e pesquisa para memória externa.

14.1.1 Gerenciamento de memória hierárquica

A maioria dos algoritmos não é projetada visando à memória hierárquica, apesar da grande variação nos tempos de acesso nos diferentes níveis. Na verdade, todas as análises de algoritmos descritas neste livro assumem que o tempo de acesso à memória é sempre igual. Essa premissa pode parecer, a princípio, omissão – e estamos tratando do tema apenas no último capítulo –, mas existem duas razões fundamentais que justificam essa escolha.

A primeira razão é que em certas situações é necessário assumir que todos os acessos à memória consomem o mesmo tempo, uma vez que as informações dependentes de dispositivo sobre os tamanhos das memórias costumam ser difíceis de obter. Na verdade tal tipo de informação pode ser impossível de conseguir. Por exemplo, um programa em Java é projetado para rodar em diferentes plataformas computacionais que não podem ser definidas em termos de uma configuração computacional específica. Certamente podemos usar informações específicas de uma arquitetura se dispusermos das mesmas (e vamos mostrar como explorar tais informações mais adiante neste capítulo). Mas uma vez que otimizamos nosso *software* para uma arquitetura específica, nosso *software* não será mais independente de dispositivo. Felizmente, tais otimizações não são sempre necessárias, basicamente por causa da segunda razão pela qual se assume que o tempo de acesso à memória é constante.

A segunda razão para assumir que os tempos de acesso à memória são sempre iguais é que os projetistas de sistema operacional têm desenvolvido mecanismos genéricos que permitem que a maioria dos acessos sejam rápidos. Esses mecanismos são baseados em duas propriedades importantes de ***localização das referências*** que a maioria dos *softwares* possui:

- ***Localização temporal*:** se um programa acessa uma certa localização na memória, então é provável que venha acessá-la novamente em um futuro próximo. Por exemplo, é comum usar o valor de uma variável contadora em várias expressões diferentes, incluindo uma para incrementá-lo. Na verdade, um ditado popular entre arquitetos de computadores diz que "um programa gasta 90% do seu tempo em 10% do seu código".
- ***Localização espacial*:** se um programa acessa uma certa localização na memória, então é possível que venha acessar outras próximas a esta. Por exemplo, um programa que usa um arranjo provavelmente irá acessá-lo de forma seqüencial ou semelhante.

Engenheiros e cientistas da computação têm realizado experimentos de *software* extensivos para justificar o clamor de que a maioria dos *softwares* possui estes dois tipos de localização de referências. Por exemplo, um laço **para** usado para varrer um arranjo irá exibir as duas formas de localização.

Cache e blocos

As localizações espacial e temporal levam, por sua vez, a duas decisões de projeto fundamentais para sistemas de computação com dois níveis de memória (que estão presentes na interface entre a memória *cache* e a memória interna e entre a memória interna e a memória externa).

A primeira decisão é chamada de ***memória virtual***. Esse conceito consiste em prover um espaço de endereçamento maior do que a capacidade da memória do segundo nível e transferir para a memória de primeiro nível dados contidos na memória de segundo nível, quando eles são endereçados. A memória virtual não limita o programador às restrições de tamanho da memória interna. O conceito de trazer dados para a memória primária é chamado de ***cache*** e é motivado pela localização temporal. Assim, ao trazermos dados para a memória primária, esperamos que estes sejam acessados novamente em breve e estaremos prontos para responder com rapidez às solicitações a respeito destes dados em um futuro próximo.

A segunda decisão de projeto é motivada pela localização espacial. Mais especificamente, se os dados armazenados na localização l da memória de segundo nível forem acessados, então trazemos para a memória de primeiro nível um grande bloco de localizações contíguas que incluam a localização l (veja a Figura 14.2). Esse conceito é conhecido como ***bloco*** e é motivado pela expectativa de que outras localizações próximas a l sejam acessadas em breve. Na interface entre a memória *cache* e a memória interna, esses blocos são normalmente chamados de ***linhas de cache***, e na interface entre a memória interna e a memória externa, são chamados de ***páginas***.

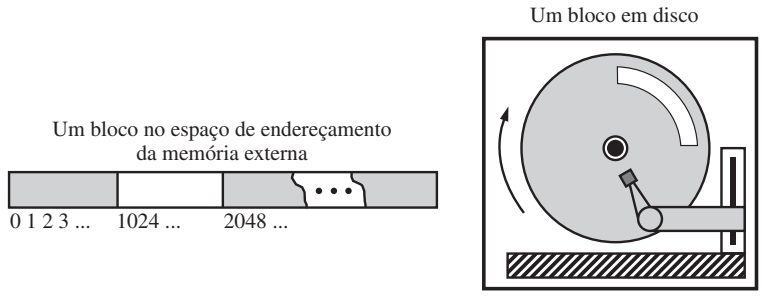


Figura 14.2 Blocos na memória externa

Casualmente, os blocos de unidades de disco e CD-ROM também são motivados pelas propriedades destas tecnologias de *hardware*. O braço de uma unidade de disco ou CD-ROM consome um tempo relativamente longo para se posicionar quando tem de efetuar a leitura de uma certa localização, mas, depois de posicionado, consegue ler rapidamente posições contíguas, porque a mídia que está lendo gira com muita velocidade (veja a Figura 14.2). Mesmo sem essa motivação, entretanto, os blocos são plenamente justificáveis, dada a propriedade de localização espacial que a maioria dos programas possui.

Assim, quando implementada com *cache* e blocos, a memória virtual normalmente nos permite perceber a memória de segundo nível como muito mais rápida do que realmente é. Entretanto, ainda existe um problema: a memória de primeiro nível é muito menor do que a memória de segundo nível. Além disso, como os sistemas de memória usam blocos, qualquer programa significativo chegará a um ponto em que irá requisitar dados da memória de segundo nível – só que a memória de primeiro nível estará cheia de blocos. Neste caso, para atender à requisição e manter o uso de *cache* e blocos, teremos de remover um bloco da memória primária para abrir espaço para o novo bloco originário da memória de segundo nível. Decidir como fazer esse despejo nos leva a uma série de projetos de algoritmos e estruturas de dados interessantes que serão discutidos no restante desta seção.

Um modelo para pesquisa externa

O primeiro problema que iremos estudar será o da implementação de um dicionário ordenado para uma grande coleção de itens que não cabem na memória principal. Lembramos que um dicionário armazena pares elemento-chave (itens) sujeitos a inserções, remoções e pesquisas baseadas em chaves. Uma vez que uma das principais aplicações dos grandes dicionários é em sistemas de banco de dados, referimo-nos a esses blocos de memória secundária como ***blocos de disco***. Da mesma forma, nos referimos à transferência de blocos entre a memória secundária e a memória principal como ***transferência de disco***. Mesmo usando essa terminologia, as técnicas de pesquisa que discutiremos nesta seção aplicam-se também quando a memória principal for a *cache* da CPU e a memória secundária for a memória principal (interna). Usamos a abordagem baseada em disco porque é concreta e mais comum.

14.1.2 Árvores (a, b) e árvores-B

Lembrando a grande diferença de tempo entre os acessos à memória principal e os acessos a disco, o principal objetivo de se manter um dicionário em memória externa é minimizar o número de transferências de disco necessárias para executar uma operação de consulta ou atualização. De fato, a diferença de velocidade entre o disco e a memória interna é tão grande que preferimos executar um número considerável de acessos à memória interna se isso nos permitir evitar algumas transferências de disco. Vamos, então, analisar o desempenho de implementações de dicionário pela contagem do número de transferências de disco que cada uma requer para executar as operações padrão de pesquisa e atualização.

Consideremos primeiramente algumas implementações ineficientes de dicionário que usam seqüências. Se a seqüência for implementada como uma lista duplamente encadeada não-ordenada, então inserções podem ser executadas com transferências $O(1)$, assumindo que conhecemos qual bloco contém o item a ser removido. Mas, neste caso, as pesquisas requerem transferências $\Theta(n)$, no pior caso, uma vez que cada ligação que executamos pode acessar um bloco diferente. Esse tempo de pesquisa pode ser melhorado para $O(n/B)$ transferências (veja o Exercício C-14.1), onde B denota o número de nodos da lista que cabem em um bloco – mas ainda é um desempenho pobre. Podemos, em vez disso, implementar a seqüência usando um vetor ordenado, isto é, uma tabela de pesquisa. Neste caso, uma pesquisa executa $O(\log_2 n)$ transferências usando algoritmos de pesquisa binária, o que é uma pequena melhoria. Porém, esta solução requer $\Theta(n/B)$ transferências para implementar uma operação de inserção ou remoção no pior caso, no qual teremos de acessar todos os blocos que armazenam o arranjo para mover os elementos para cima ou para baixo. Desta forma, implementações baseadas em seqüências de um dicionário não são eficientes do ponto de vista de memória externa.

Se implementações que usam seqüências são ineficientes, então talvez possamos considerar as estratégias com tempo logarítmico que usamos com as árvores binárias balanceadas (como as árvores AVL ou as árvores vermelho-pretas) ou outras estruturas de pesquisa com tempo médio para atualização e pesquisa logarítmica (tais como *skip lists* ou *splay trees*). Esses métodos armazenam os itens do dicionário nos nodos de uma árvore binária ou de um grafo. No pior caso, cada nodo acessado em uma consulta ou atualização em uma dessas estruturas se encontra em um bloco diferente. Sendo assim, esses métodos normalmente requerem $O(\log_2 n)$ transferências para executar uma operação de consulta ou atualização. Isso é bastante bom, mas podemos fazer melhor. Especificamente podemos executar consultas e atualizações em um dicionário usando apenas $O(\log_B n) = O(\log n/\log B)$ transferências.

Uma abordagem melhor

A idéia para melhorar o desempenho da implementação em memória externa de dicionário discutida anteriormente é que desejamos executar mais de $O(B)$ acessos à memória interna visando a evitar uma única transferência de disco, onde B denota o tamanho de um bloco de disco. O *hardware* e o *software* que controlam o disco executam esses vários acessos apenas para trazer um bloco para a memória interna e, mesmo assim, esta é uma pequena parte do custo de uma transferência de disco. Desta forma, $O(B)$ acessos à memória interna de alta velocidade são um preço pequeno para evitar uma transferência de disco demorada.

Árvores (a, b)

Para reduzir a importância da diferença de desempenho entre acessos à memória interna e acessos à memória externa em pesquisas, podemos representar nosso dicionário usando uma árvore de pesquisa genérica (Capítulo 3). Essa abordagem leva a uma generalização da estrutura de dados de árvore (2, 4) para uma estrutura conhecida como árvore (a, b).

Formalmente uma árvore (a, b) é uma árvore de pesquisa genérica tal que, em cada nodo, entre os filhos a e b, são armazenados entre $a - 1$ e $b - 1$ itens. Os algoritmos de pesquisa, inserção e remoção de elementos em uma árvore (a, b) são generalizações das operações correspondentes para árvores (2, 4). A vantagem de generalizar árvores (2, 4) em árvores (a, b) é que a classe de árvores generalizadas oferece uma estrutura de pesquisa mais flexível, em que o tamanho dos nodos e o tempo de execução das várias operações sobre o dicionário dependem dos parâmetros a e b. Ajustando os parâmetros a e b adequadamente em relação ao tamanho dos blocos de disco, podemos derivar uma estrutura de dados que obtenha um bom desempenho em memória externa.

Uma ***árvore*** (a, b), onde a e b são inteiros, tal que $2 \leq a \leq (b + 1)/2$, é uma árvore de pesquisa genérica T com as seguintes restrições adicionais:

Propriedade de tamanho: cada nodo interno tem pelo menos a filhos, a menos que seja a raiz, e no máximo b filhos.

***Propriedade de profundidade*:** todos os nodos externos têm a mesma profundidade.

Teorema 14.1: *a altura de uma árvore (a, b) que armazena n itens é* $\Omega(\log n/\log b)$ *e* $O(\log n/\log a)$.

Prova: *seja T uma árvore (a, b) que armazena n elementos e seja h a altura de T. Justificamos a proposição estabelecendo os seguintes limites para h:*

$$\frac{1}{\log b}\log(n+1) \leq h \leq \frac{1}{\log a}\log\frac{n+1}{2} + 1.$$

Pelas propriedades de tamanho e profundidade, o número n^n de nodos externos de T é no mínimo $2a^{h-1}$, e no máximo b^h. Pelo Teorema 3.3, $n^n = n + 1$. Então,

$$2a^{h-1} \leq n + 1 \leq b^h.$$

Tomando o logaritmo de base 2 para cada termo, obtemos

$$(h - 1) \log a + 1 \leq \log(n + 1) \leq h \log b.$$

■

Lembramos que, em uma árvore de pesquisa genérica T, cada nodo v de T armazena uma estrutura secundária $D(v)$, que é também um dicionário (Seção 3.3.1). Se T for uma árvore (a, b), então $D(v)$ armazena no máximo b itens. Faça $f(b)$ denotar o tempo para executar uma pesquisa em um dicionário $D(v)$. O algoritmo de pesquisa em uma árvore (a, b) é exatamente igual ao de árvores de pesquisa genéricas apresentado na Seção 3.3.1. Logo, pesquisar em uma árvore (a, b) T com n itens leva tempo $O\left(\frac{f(b)}{\log a}\log n\right)$.

Observamos que, se b é constante (e conseqüentemente a também é), então o tempo de pesquisa é $O(\log n)$, independentemente da implementação específica das estruturas secundárias.

A principal aplicação de árvores (a, b) é em dicionários armazenados em memória externa (por exemplo, em um disco ou CD-ROM). De fato, para minimizar os acessos a disco, selecionamos os parâmetros a e b de maneira que cada nodo da árvore ocupe um único bloco de disco (então $f(b) = 1$, se desejamos simplesmente contar as transferências de bloco). Prover os valores corretos para a e b neste contexto leva à geração de uma estrutura conhecida como árvore-B, que descreveremos resumidamente. Antes de analisarmos essa estrutura, entretanto, discutiremos como as inserções e remoções são tratadas em árvores (a, b).

Inserção e remoção em uma árvore (a, b)

O algoritmo de inserção para uma árvore (a, b) é similar ao de uma árvore (2,4). Um *overflow* ocorre quando um item é inserido em um b-nodo v, que se transforma em um nodo ilegal $(b + 1)$-nodo. (Lembramos que um nodo em uma árvore de pesquisa genérica é um ***d*-nodo** se tiver d filhos.) Para remediar um *overflow*, dividimos o nodo v movendo o item médio de v para seu pai e substituindo v por um $\lceil (b + 1)/2 \rceil$-nodo v' e por um $\lfloor (b + 1)/2 \rfloor$-nodo v''. Podemos ver agora a razão para requerer que $a \leq (b + 1)/2$ na definição de uma árvore (a, b). Observamos que, como conseqüência da divisão, necessitamos construir as estruturas secundárias $D(v')$ e $D(v'')$.

Remover um elemento de uma árvore (a, b) também é similar ao que foi feito para árvores (2,4). Um *underflow* ocorre quando uma chave é removida de um a-nodo v, diferente da raiz, que faz com que v se transforme em um nodo ilegal $(a - 1)$-nodo. Para remediar um *underflow*, executamos ou uma transferência com um irmão de v que não seja um a-nodo, ou a fusão de v com um irmão que seja um a-nodo. O novo nodo w resultante da fusão é um $(2a - 1)$-nodo. Aqui vemos outra razão para requerer que $a \leq (b + 1)/2$. Observamos que, como conseqüência da fusão, necessitamos construir a estrutura secundária $D(w)$.

A Tabela 14.3 apresenta o tempo de execução das principais operações de um dicionário implementado usando uma árvore (a, b) T.

Método	**Tempo**
findElement	$O\left(\frac{f(b)}{\log a}\log n\right)$
insertItem	$O\left(\frac{f(b)}{\log a}\log n\right)$
removeElement	$O\left(\frac{f(b)}{\log a}\log n\right)$

Tabela 14.3 Complexidade de tempo dos principais métodos de um dicionário implementado por uma árvore (a,b). Fazemos $f(b)$ denotar o tempo de pesquisa de um nodo-b e $g(b)$, o tempo para dividir ou fundir um nodo-b. Também denotamos o número de elementos do dicionário por n. A complexidade do espaço de uso é $O(n)$.

Os limites de tempo da Tabela 9.5 são baseados nas seguintes premissas e fatos:

- A árvore (a, b) T é implementada usando a estrutura de dados descrita na Seção 3.3.1, e a estrutura secundária dos nodos de T suporta pesquisa em tempo $f(b)$ e operações de divisão e fusão em tempo $g(b)$ para algumas funções $f(b)$ e $g(b)$ que podem ser feitas em $O(1)$, se contarmos apenas as transferências de disco.
- A altura de uma árvore (a, b) que armazena n elementos é no máximo $O(\log n)/(\log a))$ (Teorema 14.1).
- Uma pesquisa visita $O(\log n)/(\log a))$ nodos no caminho entre a raiz e um nodo externo e consome tempo $f(b)$ por nodo.
- Uma operação de transferência leva tempo $f(b)$.
- Uma operação de divisão ou fusão leva tempo $g(b)$ e exige uma estrutura secundária de tamanho $O(b)$ para os novos nodos criados.
- A inserção ou remoção de um elemento visita $O((\log n)/(\log a))$ nodos no caminho entre a raiz e o nodo externo e gasta tempo $g(b)$ por nodo.

Então, podemos resumir como segue.

Teorema 14.2: *uma árvore (a, b) implementa um dicionário de n itens que suporta a execução de inserções e remoções em tempo* $O((g(b)/\log a)\log n))$, *e executa operações de consulta em tempo* $O((f(b)/\log a)\log n)$.

Árvores-B

Uma versão especializada da estrutura de dados árvore (a, b), que é um método eficiente para manutenção de dicionários em memória externa, é conhecida como "árvore-B" (veja a Figura 14.4). ***Uma árvore-B de ordem d*** é uma árvore (a, b) com $a = \lceil d/2 \rceil$ e $b = d$. Como já discutimos anteriormente as operações-padrão para consulta e atualização de dicionários usando árvores (a, b), restringiremos nossa discussão à análise de *performance* em memória externa de árvores-B.

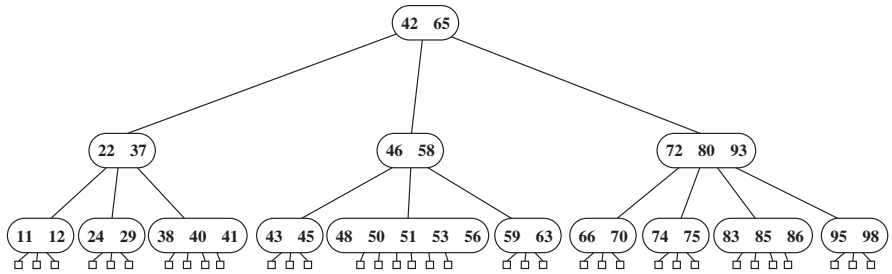


Figura 14.4 Uma árvore-B de ordem 6.

Parametrização de árvores-B para memória externa

A observação mais importante sobre as árvores-B é que podemos escolher d de maneira que as d referências para filhos e as $d - 1$ chaves armazenadas em um nodo possam todas caber em um único bloco de disco. Ou seja, escolhemos d de maneira que

$$d \text{ seja } \Theta(B).$$

Essa escolha também nos permite assumir que a e b são $\Theta(B)$ na análise das operações de pesquisa e atualização em árvores (a, b). Lembre também que estamos interessados primeiramente no número de transferências de disco necessárias para executar várias operações. Assim, a escolha de d também implica que

$$f(b) = c,$$

e

$$g(b) = c,$$

para uma constante $c \geq 1$, pois cada vez que acessamos um nodo para executar uma operação de pesquisa ou atualização, necessitamos apenas de uma única transferência de disco. Ou seja, $f(b)$ e $g(b)$ são $O(1)$. Como mencionamos antes, cada pesquisa ou atualização requer que examinemos no mínimo $O(1)$ nodos para cada nível da árvore. Conseqüentemente, qualquer operação de pesquisa ou atualização em uma árvore-B requer apenas

$$\begin{aligned} O(\log_{\lceil d/2 \rceil} n) &= O(\log n / \log B) \\ &= O(\log_B n) \end{aligned}$$

transferências de disco. Por exemplo, uma operação de inserção percorre a árvore para baixo visando a localizar o nodo no qual insere-se um novo item. Se o nodo ***overflow*** (ficar com $d + 1$ filhos) em função desta adição, então este nodo é ***dividido*** em dois que terão $\lfloor (d + 1)/2 \rfloor$ e $\lceil (d + 1)/2 \rceil$ filhos, respectivamente. Esse processo é então repetido no nível de cima e irá continuar por pelo menos $O(\log_B n)$ níveis.

Da mesma forma, em uma operação de remoção, retiramos um item de um nodo, e se isso resultar em um ***underflow*** de um nodo (com $\lceil (d/2) - 1 \rceil$ filhos), então moveremos as referências de um nodo irmão com pelo menos $\lceil (d/2) + 1 \rceil$ filhos ou necessitaremos executar uma operação de ***fusão*** deste nodo com seu irmão (e repetir esta operação para seu pai). Da mesma forma, com a operação de inserção, isso irá continuar pela árvore-B acima por pelo menos $O(\log_B n)$ níveis. Assim, teremos o seguinte:

Teorema 14.3: *uma árvore-B com n itens executa $O(\log_B n)$ transferências de disco para operações de pesquisa ou atualização, onde B é o número de itens que cabem em um bloco.*

O requisito de que cada nodo interno tenha pelo menos $\lceil (d/2) \rceil$ filhos implica que cada bloco de disco usado para suportar uma árvore-B esteja pelo menos completo pela metade. Um estudo analítico e experimental da média de uso dos blocos em uma árvore-B diz que esta é próxima de 67%, o que é muito bom.

14.1.3 Ordenação em memória externa

Além das estruturas de dados, tais como os dicionários que necessitam ser implementadas em memória externa, existem vários algoritmos que têm de operar com conjuntos de entrada muito grandes para caber inteiramente na memória interna. Neste caso, o objetivo é resolver o problema algorítmico usando a menor quantidade de transferências de blocos possível. O domínio mais clássico deste tipo de algoritmo de memória externa é o problema de ordenação.

Limite inferior para ordenação em memória externa

Como discutimos antes, pode existir uma grande diferença no desempenho de um algoritmo se executado em memória interna ou externa. Por exemplo, o desempenho do algoritmo *radix-sort* é péssimo em memória externa, ainda que seja muito bom em memória interna. Outros algoritmos, tais como o *merge-sort*, são razoavelmente bons em ambas. O número de transferências de blocos executadas pelo algoritmo de *merge-sort* tradicional é $O((n/B)\log_2 n)$, onde B é o tamanho dos blocos do disco. Apesar de isso ser muito melhor que a quantidade de transferências $O(n)$ executada pela versão para memória externa do *radix-sort*, não é, de forma nenhuma, o melhor que podemos obter para o problema da ordenação. Na verdade podemos demonstrar o seguinte limite inferior, cuja prova está fora do escopo deste livro.

Teorema 14.4: *ordenar n elementos em memória externa requer*

$$\Omega\left(\frac{n}{B}\cdot\frac{\log(n/B)}{\log(M/B)}\right)$$

transferências de bloco, onde M é o tamanho da memória interna.

A razão M/B é o número de blocos de memória externa que cabem na memória interna. Assim, esse teorema está dizendo que o melhor desempenho que podemos obter para o problema de ordenação é equivalente ao trabalho de varrer o conjunto de entrada (o que consome $\Theta(n/B)$ transferências) pelo menos um número logarítmico de vezes, onde a base deste logaritmo é o número de blocos que cabem na memória interna. Não iremos justificar formalmente esse teorema, mas mostraremos como projetar algoritmos de ordenação para memória externa cujos tempos de execução sejam um fator constante deste limite inferior.

Merge-sort genérico

Uma forma eficiente de ordenar um conjunto S de n objetos em memória externa é uma variação simples do algoritmo *merge-sort* tradicional. A idéia principal por trás dessa variação é unir várias listas ordenadas recursivamente de cada vez, de maneira a reduzir o número de níveis de recursão. Mais especificamente, uma descrição de alto nível deste método de ***merge-sort genérico*** é dividir S em d subconjuntos $S_1, S_2,..., S_d$ de tamanho aproximadamente iguais, ordenar recursivamente cada subconjunto S_i, e então unir simultaneamente todas as d listas ordenadas em uma representação ordenada de S. Se executarmos o processo de junção usando apenas $O(n/B)$ transferências de disco, então, para valores grandes o suficiente de n, o número total de transferências executadas por esse algoritmo satisfaz à seguinte repetição:

$$t(n) = d \cdot t\,(n/d) + cn/B,$$

para alguma constante $c \geq 1$. Podemos parar a recursão quando $n \leq B$, uma vez que podemos executar uma única transferência de bloco neste ponto, tomando todos os objetos da memória interna e, então, ordenando o conjunto com um algoritmo eficiente para memória interna. Assim, o critério de parada para $t(n)$ é

$$t(n) = 1 \qquad \text{se } n/B \leq 1.$$

Isso implica uma solução de forma fechada, onde $t(n)$ é $O((n/B)\log_d(n/B))$, que é

$$O((n/B)\log\,(n/B)/\log d).$$

Assim, se pudermos escolher d para ser $\Theta(M/B)$, então o pior caso em relação à quantidade de transferências de blocos executadas pelo algoritmo *merge-sort* genérico será um fator constante no limite mínimo dado no Teorema 14.4. Escolhemos

$$d = (1/2)M/B.$$

O único aspecto desse algoritmo que ainda ficou por especificar, então, é como executar a junção d-formas usando apenas $O(n/B)$ transferências de bloco.

Executamos a junção d-forma executando um "torneio". Fazemos T ser uma árvore binária completa com d nodos externos e mantemos T inteiro na memória interna. Associamos cada nodo externo i de T com uma lista ordenada diferente S_i. Inicializamos T lendo para cada nodo externo i, o primeiro objeto de S_i. Isto tem o efeito de ler na memória interna o primeiro bloco de cada lista ordenada S_i. Para cada nodo interno pai v de dois nodos externos, comparamos os objetos armazenados nos filhos de v e associamos o menor dos dois a v. Repetimos esta comparação, então, no nível imediatamente acima em T e no próximo, e assim por diante. Quando alcançarmos a raiz de T, iremos associar o menor objeto entre os de todas as listas com r. Isso completa a inicialização da junção d-forma (veja a Figura 14.5).

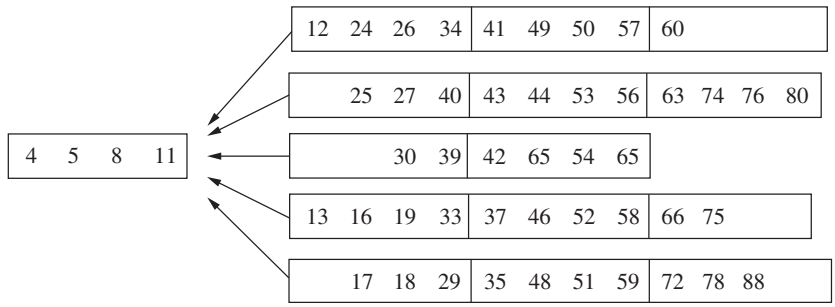


Figura 14.5 Junção d-forma. Apresentamos uma junção 5-forma com $B = 4$.

Em um passo normal de uma junção d-forma, movemos o objeto o associado com a raiz r de T para um arranjo que estamos construindo para a lista unida S'. Quando percorremos T para baixo, seguimos o caminho para o nodo externo i de onde o se origina. Lemos então, de i, o próximo objeto na lista S_i. Se o não for o último elemento deste bloco, então o próximo objeto já está na memória interna. Caso contrário, lemos o próximo bloco de S_i para acessar o novo objeto (se S_i estiver vazio, associa-se com o nodo i um pseudo-objeto com chave $+\infty$). Repetimos então os cálculos mínimos para cada um dos nodos internos partindo de i até a raiz de T. Isso nos fornece novamente uma árvore completa T. Repetimos então o processo de mover o objeto da raiz de T para a lista unida S' e reconstruímos T até ficar sem nenhum objeto. Cada passo da junção consome tempo $O(\log d)$; conseqüentemente, o tempo interno para uma junção d-forma é $O(n \log d)$. A quantidade de transferências executadas em uma junção é $O(n/B)$, já que varremos uma vez cada lista S_i em ordem e escrevemos uma vez a lista resultante S'. Então temos:

Teorema 14.5: *dada uma seqüência S baseada em um arranjo S de n elementos armazenados em uma memória externa, podemos ordenar S usando $O((n/B)\log(n/B)/\log(M/B))$ transferências e $O(n \log n)$ tempo interno de CPU, onde M é o tamanho da memória interna e B é o tamanho do bloco.*

Obtendo uma "quase" independência de máquina

Usar árvores-B e algoritmos de ordenação externos pode produzir reduções significativas no número de transferências de blocos. A informação mais importante que torna essas reduções possíveis é conhecer o valor de B, o tamanho de um bloco de disco (ou linha de *cache*). Essa informação é, obviamente, dependente de máquina, mas é uma das informações menos dependentes dentre as necessárias – uma delas é a habilidade de armazenar chaves em arranjos de forma contínua.

A partir de nossa descrição de árvores-B e ordenação externa, podemos ser levados a acreditar que necessitamos de acesso de baixo nível ao gerenciador de dispositivo da memória externa, mas isso não é obrigatório para a obtenção dos resultados declarados em relação a um fator constante. Em particular, além de conhecer o tamanho do bloco, a única outra coisa que precisamos saber é se grandes arranjos de chaves são particionados em blocos de células contínuas. Isso nos permite implementar os "blocos" nas árvores-B e nosso algoritmo de ordenação de memória externa como arranjos separados de tamanho B, que chamamos de **pseudoblocos**. Se os arranjos são alocados em blocos da forma natural, qualquer pseudobloco será alocado em pelo menos dois blocos reais. Assim, mesmo que deixemos que o sistema operacional faça a substituição de blocos (por exemplo, usando FIFO, LRU ou a política de marcação que será discutida na Seção 14.3), podemos estar certos de que acessar qualquer pseudobloco requer no máximo duas, isto é, $O(1)$ transferências de bloco reais. Usando pseudoblocos em vez de blocos reais, podemos implementar o TAD dicionário para obter operações de pesquisa e atualização que usem apenas $O(\log_B n)$ transferências de bloco. Podemos, assim, projetar estruturas de dados e algoritmos para memória externa sem ter controle completo da hierarquia de memória do sistema operacional.

14.2 Algoritmos paralelos

Nesta seção, discutiremos algoritmos paralelos e algumas técnicas fundamentais de algoritmos paralelos, incluindo divisão e conquista paralela simples, subconjuntos seqüenciais, teorema de Brent, duplicação recursiva e prefixo paralelo e junção e ordenação paralelos. Todas essas técnicas têm provado ser úteis no projeto de algoritmos paralelos eficientes. Concluímos esta seção apresentando uma aplicação de algumas dessas técnicas no problema de encontrar o diâmetro de um polígono convexo.

14.2.1 Modelos paralelos de computação

Estender o modelo RAM para permitir múltiplos processadores leva a um modelo paralelo conhecido como RAM Paralela ou ***PRAM*** (*Parallel RAM*). Este é um modelo paralelo síncrono no qual todos os processadores compartilham uma memória comum. Devido à sua simplicidade conceitual, este modelo é parecido com o modelo que quase todos os computadores paralelos emulam de uma forma ou de outra. O modelo PRAM também é adequado para descobrir o paralelismo inerente que pode estar presente em um problema. Finalmente, o modelo PRAM parece ser apropriado para determinar paradigmas genéricos que podem ser usados para desenvolver algoritmos paralelos eficientes.

Eficiência paralela

No ambiente seqüencial, dizemos que um algoritmo é "bom" se tiver um tempo de execução que seja no máximo $O(n^k)$ sendo k constante, isto é, execute em tempo polinomial. Como referido no Capítulo 13, um problema é considerado "tratável" se existir um algoritmo polinomial capaz de resolvê-lo. A noção correspondente em um ambiente paralelo é dizer que um algoritmo é bom se seu tempo de execução for $O(\log^{k_1} n)$ usando $O(n^{k_2})$ processadores, onde k_1 e k_2 são constantes, ou seja, ele executa em tempo polilogarítmico usando um número polinomial de processadores. De forma análoga, no jargão de complexidade de algoritmos, diz-se que um problema pertence à classe *NC* se existir um algoritmo que o resolva que seja bom neste sentido.

No ambiente seqüencial, uma vez que se sabe que o problema pode ser resolvido em tempo polinomial, o objetivo passa a ser encontrar o algoritmo que resolve o problema o mais rápido possível. Da mesma forma, no ambiente paralelo, uma vez que se sabe que o problema é *NC*, o objetivo passa a ser encontrar um algoritmo que resolva o problema e minimize o produto $T(n) * P(n)$ (no sentido assintótico), onde $T(n)$ é a complexidade temporal do algoritmo, e $P(n)$ é o número de processadores usados pelo mesmo. Ou seja, se $Seq(n)$ denotar o tempo de execução seqüencial para resolver um certo problema, então queremos que $T(n) * P(n)$ esteja o mais próximo possível de $Seq(n)$, no sentido assintótico. Esse objetivo é motivado pelo fato de que um único processador pode simular os cálculos de $P(n)$ processadores executando um passo de cada um em uma abordagem *round-robin* (veja a Seção 2.1.2).

Dizemos que um algoritmo paralelo é ***ótimo*** se o produto $T(n) * P(n)$ combinar com o limite inferior seqüencial do problema que resolve. Tecnicamente essa definição permite que um algoritmo seqüencial (com $P(n) = 1$) seja também um algoritmo paralelo. Assim, dado que $T(n) * P(n)$ esteja próximo de $Seq(n)$ para algum problema, o segundo objetivo passa a ser minimizar $T(n)$, o tempo de execução. A motivação para esses dois objetivos é que qualquer máquina existente, com digamos k processadores (uma constante), pode simular um algoritmo que seja eficiente neste sentido, de maneira a obter o maior aumento de velocidade. Os k processadores podem simular cada passo dos $P(n)$ processadores em tempo $\lceil P(n)/k \rceil$, sendo que cada um dos k processadores executam o trabalho de $\lceil P(n)/k \rceil$ processadores. Essa abordagem resulta em um algoritmo que executa em tempo $O(T(n)P(n)/k + T(n))$. Além disso, qualquer limite inferior para a execução seqüencial necessária para resolver um certo problema imediatamente se torna um limite inferior para o produto $T(n) * P(n)$ (porque uma máquina seqüencial pode simular um algoritmo PRAM com esses limites em tempo $O(T(n) * P(n))$.

Versões do modelo PRAM

Existem basicamente três versões diferentes para o modelo PRAM, cada uma diferindo da outra na forma de resolver conflitos de memória. (Lembramos que em todos os modelos PRAM os processadores operam de forma síncrona.) A versão mais restritiva é a de LeituraExclusiva/EscritaExclusiva (ou EREW – *Exclusive-Read, Exclusive-Write*) PRAM, na qual não são permitidas leituras ou escritas simultâneas. Se permitirmos que mais de um processador execute leituras simultâneas, mas ainda restringirmos as escritas simultâneas, teremos o LeituraConcorrente/EscritaExclusiva (ou CREW – *Concurrent-Read, Exclusive-Write*) PRAM. Finalmente, se permitirmos que mais de um processador escreva simultaneamente na mesma célula de memória,

então teremos o modelo mais poderoso, o LeituraConcorrente/EscritaConcorrente (ou CRCW – *Concurrent-Read, Concurrent-Write*) PRAM. O método de resolução de conflito varia, mas os dois métodos mais comuns são baseados na definição de um modelo de acordo com as seguintes regras:

- Restringir todos os processadores que estiverem escrevendo na mesma posição a escreverem o mesmo valor.
- Quando $k > 1$ processadores estão escrevendo simultaneamente na mesma posição, permita que apenas um destes processadores tenha sucesso (arbitrariamente).

Como exemplo de algoritmo CRCW PRAM, suponha que temos um arranjo A de booleanos e solicitamos o cálculo de um OR em todos os *bits* de A. Podemos inicializar o *bit* de saída o para 0 e atribuir um processador a cada *bit* de A. Então podemos ter cada processador atribuído a um *bit* que seja 1 ou escrever o valor 1 em o. Esse algoritmo simples nos permite calcular o OR de n *bits* em tempo $O(1)$ usando n processadores no modelo CRCW PRAM (usando as regras acima para resolver o conflito).

Esse exemplo simples mostra o poder do modelo CRCW PRAM, que, entretanto, não é considerado realista por muitos pesquisadores. Assim, todos os outros algoritmos descritos nesta seção serão para os modelos CREW PRAM e EREW PRAM.

14.2.2 Divisão e conquista paralela simples

A técnica paralela de divisão e conquista é análoga à técnica seqüencial de divisão e conquista. A abordagem é a seguinte: dado um problema, o dividimos em uma pequena quantidade de subproblemas, similares em estrutura ao problema original, mas menores em tamanho, e então resolvemos cada subproblema recursivamente em paralelo. De maneira a contar com um algoritmo eficiente, devemos ter a habilidade de unir as soluções dos subproblemas rapidamente em paralelo à medida que as chamadas recursivas em paralelo retornam.

Como um exemplo fácil de uso desta técnica, consideremos o problema de calcular a soma de n inteiros (a mesma técnica funciona para qualquer operação associativa). Vamos assumir que a entrada seja um arranjo contendo n inteiros e já resida na memória. Por simplicidade, vamos assumir que n seja potência de dois. Dividimos o arranjo em dois subarranjos contendo $n/2$ inteiros cada e encontramos a soma dos inteiros de cada um, recursivamente em paralelo. Depois que a chamada recursiva retornar, podemos calcular a soma de todos os n inteiros pela adição das somas calculadas para cada subarranjo. Uma vez que isso pode ser feito em tempo $O(1)$, a complexidade temporal $T(n)$ satisfaz a equação recorrente $T(n) = T(n/2) + c$, o que implica que $T(n) = O(\log n)$. O número de processadores $P(n)$ necessários para esse cálculo pode ser expresso na equação recorrente $P(n) = \max\{2P(n/2),1\}$, que tem solução $P(n) = O(n)$. Observamos que o produto $T(n) * P(n)$, neste caso, é $O(n \log n)$, que está distante do ótimo de um fator $\log n$. Na próxima subseção, discutiremos dois métodos que podem ser usados para reduzir o número de processadores de maneira que o produto $T(n) * P(n)$ seja ótimo, ou seja, $O(n)$.

14.2.3 Subconjuntos seqüenciais e o teorema de Brent

Tanto a técnica dos ***subconjuntos seqüenciais*** como o ***teorema de Brent*** envolvem a construção de um algoritmo paralelo eficiente a partir de algoritmos existentes que não são muito efetivos por si mesmos. Começamos nossa discussão com a técnica dos subconjuntos seqüenciais. A idéia principal dessa técnica é executar uma quantidade limitada de pré-processamentos seqüenciais em pequenos subconjuntos do problema a ser resolvido, e então aplicar um algoritmo paralelo para finalizar a solução do problema. Em algumas instâncias, pode ser que haja um pós-processamento seqüencial que deva ser feito em cada subconjunto após o algoritmo paralelo se completar. Demonstraremos como essa técnica pode ser aplicada ao problema da soma dos n números. Inicialmente, podemos dividir o arranjo em $n/\log n$ subarranjos de $\log n$ elementos cada, atribuir um processador para cada subarranjo e somar os elementos de cada um seqüencialmente. Então podemos aplicar o procedimento de divisão e conquista em paralelo, descrito na subseção anterior, para somar as $n/\log n$ somas parciais recém-calculadas. A etapa de pré-processamento executa em tempo $O(\log n)$ usando $O(n/\log n)$ processadores, como cada passo da divisão e conquista em paralelo. Assim, este irá resultar em um produto $T(n) * P(n)$ ótimo de $O(n)$ para calcular a soma de n inteiros.

Teorema de Brent

O teorema de Brent também é uma técnica de redução do número de processadores necessários para resolver um problema em particular. Se um algoritmo for projetado de maneira que um grande número de processadores permaneça disponível durante a maior parte do processamento, o teorema de Brent pode ser usado. Esse teorema pode ser resumido como segue.

Teorema 14.6: *qualquer algoritmo paralelo síncrono que execute em T etapas de tempo e consista em um total de N operações pode ser simulado por P processadores em um tempo que corresponde a*

$$O(\lfloor N/P \rfloor + T).$$

Prova: (esboço) seja N_i o número de operações executadas em uma etapa de tempo i do algoritmo paralelo. Os P processadores podem simular a etapa i do algoritmo em tempo $O(\lceil N_i / P \rceil)$. Assim, o tempo total de execução é $O(\lfloor N/P \rfloor + T)$, uma vez que

$$\begin{aligned} \sum_{i=1}^{T} \lceil N_i / P \rceil &\leq \sum_{i=1}^{T} \left(\lfloor N_i / P \rfloor + 1 \right) \\ &\leq \lfloor N / P \rfloor + T. \end{aligned}$$

■

Existem, entretanto, duas modificações a ser feitas no teorema de Brent antes que possamos aplicá-lo no modelo PRAM. Primeiramente, devemos ser capazes de calcular N_i no início da etapa i em tempo $O(\lceil N_i / P \rceil)$ usando P processadores. Ou seja, precisamos saber quantas operações devemos realmente processar durante a etapa i. E, em segundo lugar, devemos saber exatamente como atribuir a cada processador sua tarefa. Isto é, devemos ser capazes de direcionar a cada processador as $O(\lceil N_i / P \rceil)$ operações que devem executar na etapa simulada i.

Como exemplo de aplicação do teorema de Brent, consideremos o problema do somatório novamente. Lembremo-nos do algoritmo de divisão e conquista para somatório apresentado anteriormente. Apesar de o algoritmo executar em tempo $O(\log n)$ e usar $O(n)$ processadores, executava apenas $O(n)$ operações no total. Assim, aplicando o teorema de Brent, temos um método alternativo de resolver esse problema em tempo $O(\log n)$ usando $O(n/\log n)$ processadores. As modificações no teorema de Brent são facilmente resolvidas neste caso, uma vez que o número de operações executadas na etapa i é exatamente a metade do número executado na etapa $i - 1$. Mais especificamente, consumiremos tempo $\lceil \log n \rceil$ simulando a primeira etapa (quando todos os n processadores estão ativos), $\lceil \log n/2 \rceil$ simulando a segunda etapa, $\lceil \log n/4 \rceil$ para a terceira, e assim por diante. Isso nos leva a $O(\log n)$ com clareza, fornecendo-nos um método alternativo para somar n inteiros em tempo $O(\log n)$, usando $O(n/\log n)$ processadores.

14.2.4 Recursão dupla

A recursão dupla é uma técnica que pode ser entendida como um complemento do paradigma de divisão e conquista – enquanto que divisão e conquista é uma técnica *top-down*, dupla recursividade é uma técnica *bottom-up*. A idéia básica é iniciar com pequenos conjuntos e iterativamente combiná-los em pares até termos resolvido todo o problema. Descrevemos a técnica de recursividade dupla com um exemplo: a ***lista de posições***. No problema da lista de posições fornecemos uma lista encadeada representada por um arranjo de ponteiros residente em memória e somos requisitados a calcular a distância de cada elemento do fim da lista. Discutimos aqui o algoritmo clássico para este problema. A idéia básica é atribuir a cada processador um elemento da lista, e a cada etapa de tempo, atribuir ao ponteiro de cada elemento o ponteiro de seu sucessor. Em cada iteração, a distância que este elemento é capaz de "olhar à frente" é duplicada, logo, em no máximo $O(\log n)$ iterações, cada elemento estará apontando para o fim da lista. Mas especificamente, façamos $p(v)$ denotar o ponteiro de v, e faça *tail* denotar o nodo que corresponde ao fim da lista. Armazenamos também um rótulo $r(v)$ com cada nodo v que seja inicialmente 1 para todos os nodos, exceto o último (cujo rótulo é 0). Este rótulo irá armazenar por fim a colocação de v na lista. Em cada etapa de tempo, a seguinte operação é executada para cada nodo v em paralelo (veja a Figura 14.6):

se $p(v) \neq tail$ **então**
    $r(v) := r(p(v)) + r(v)$; {fatia de colocação}
    $p(v) := p(p(v))$; {fatia de "duplicação"}

Assim, temos o seguinte:

Teorema 14.7: *podemos encontrar, para cada nodo v em uma lista de nodos L, a distância de v para o fim da lista em tempo O(log n), usando O(n) processadores no modelo CREW PRAM.*

Observamos que o produto acima, $T(n) * P(n)$, é um fator $\log n$ do ótimo.

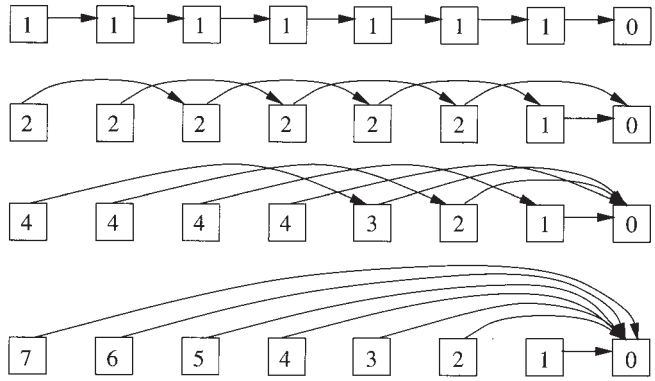


Figura 14.6 Recursão dupla aplicada ao problema da lista de colocações.

Prefixo paralelo

O problema do ***prefixo paralelo*** é um problema relacionado que também pode ser resolvido usando recursão dupla. Dada uma lista $A = (a_1, a_2, \ldots, a_n)$ de inteiros, queremos calcular a soma de todos os prefixos $s_k = \Sigma_{i=1}^{k} a_i$. Usando a técnica da recursão dupla, podemos fazê-lo facilmente em tempo $O(\log n)$, utilizando $O(n)$ processadores (veja a Figura 14.7). O método é basicamente o mesmo que empregamos para a lista de colocações.

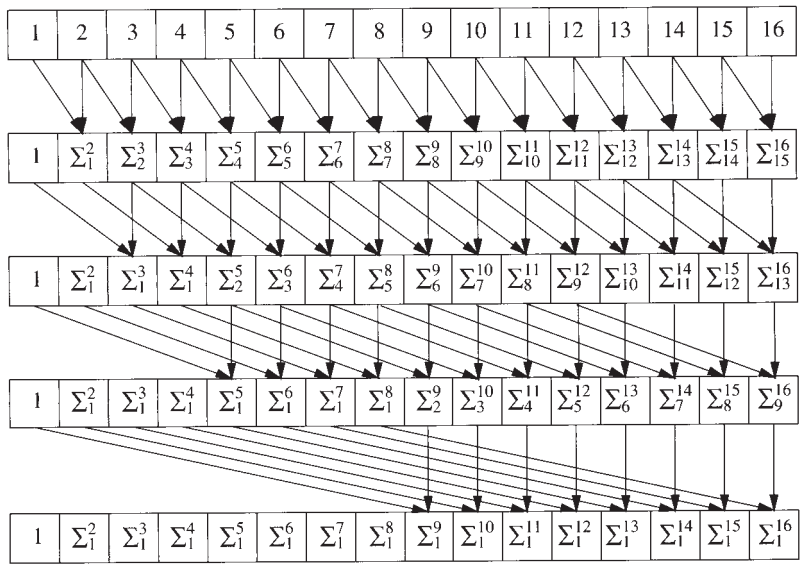


Figura 14.7 Recursão dupla aplicada ao problema do prefixo paralelo.

Ao contrário do problema das colocações, podemos aplicar, de maneira quase direta, a técnica dos subconjuntos seqüenciais ao problema do prefixo paralelo. Neste caso, podemos reduzir o número de processadores para $O(n/\log n)$ e ainda obter um tempo de execução $O(\log n)$, resultando em um produto ótimo $T(n) * P(n)$. O método para fazer isso é o seguinte: dividamos o arranjo A em $n/\log n$ subarranjos de tamanho

log n e calculemos a soma de cada subarranjo, seqüencialmente, usando um único processador. Então calculemos o prefixo como antes, usando as somas parciais como sendo os elementos. Após completar o cálculo dos prefixos destes elementos, simplesmente "retornamos", através de cada um dos n/log n subarranjos, um processador por subarranjo, para calcular todas as somas parciais (veja a Figura 14.8).

Os prefixos paralelos podem ser entendidos como um subproblema dentre outros problemas de manipulação. Demonstraremos isso através de dois exemplos.

Primeiramente, consideremos o seguinte problema de compactação de arranjo: dado um arranjo A de n-elementos no qual cada elemento é anotado como sendo "marcado" ou "não-marcado", vamos construir um arranjo B contendo os elementos marcados de A na ordem em que aparecem em A (isto é, B é a subseqüência marcada de A). O método consiste em associar o valor 1 a cada elemento marcado e o valor 0 a cada elemento não-marcado. Executando o cálculo do prefixo paralelo, podemos determinar a colocação na lista comprimida B de cada elemento marcado de A. Passa a ser uma operação simples, então, escrever cada elemento marcado de A na posição correta de B. Isso pode ser feito claramente em tempo $O(\log n)$ usando $O(n)$ processadores.

Em segundo lugar, consideremos o problema da ***divisão de um arranjo***: dado um arranjo A composto de n inteiros, vamos decompor A nos subarranjos $A_1, A_2,..., A_n$ de forma que cada A_i corresponda às repetições do mesmo inteiro, diferente do inteiro repetido em A_{i-1} e do repetido em A_{i+1} (isto é, $k \in A_i$ implica $k \notin A_{i-1}$ e $k \notin A_{i+1}$). Isso pode ser feito associando o valor 1 a cada elemento de A e executando o passo de combinação do algoritmo do prefixo paralelo para cada elemento, na medida em que todos os elementos precedentes são idênticos. Isso irá nos fornecer a colocação de cada elemento a neste subarranjo. Podemos então executar outra operação de prefixo paralelo para determinar a qual subarranjo A_i a pertence (isto é, determinar o valor de i).

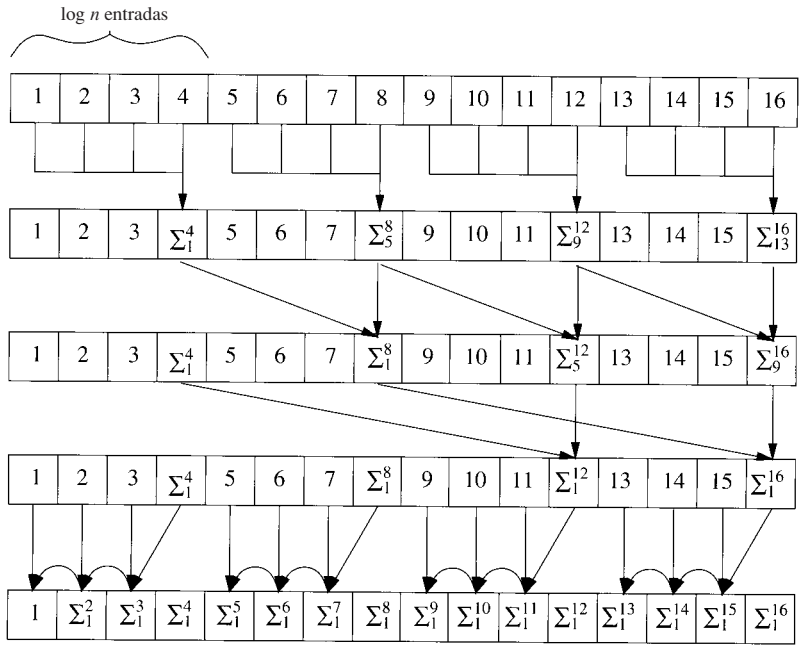


Figura 14.8 Subconjuntos seqüenciais aplicados ao problema do prefixo paralelo.

14.2.5 Junção e ordenação paralelas

Unir duas listas ordenadas ou ordenar n valores são computações fáceis de executar de forma eficiente em um modelo seqüencial, mas não triviais em paralelo. Existe um volume considerável de trabalho dedicado a esses dois problemas que trata do algoritmo de junção mais eficiente sendo capaz de unir duas listas ordenadas em tempo $O(\log n)$ usando $n/\log n$ processadores. O algoritmo mais eficiente de ordenação pode ordenar n valores em tempo $O(\log n)$ usando n processadores. Além disso, alguns desses algoritmos podem ser executados no modelo EREW PRAM com os mesmos limites assintóticos.

Infelizmente, os algoritmos de junção e ordenação em paralelo mais eficientes são muito complicados para ser indicados em detalhes aqui. Em vez disso, vamos descrever um método simples para unir duas listas ordenadas A e B de n elementos distintos em tempo $O(\log n)$ usando n processadores no modelo CREW PRAM. Para qualquer elemento x, defina a ***colocação*** de x em A ou B como a quantidade de elementos naquela lista que sejam menores ou iguais a x. Nosso algoritmo determina a colocação em B de cada item de A e a colocação em A de cada item de B. Em particular, atribuímos um processador a cada item de A e B e executamos uma pesquisa binária para determinar a posição de cada item na outra lista. Isso consome tempo $O(\log n)$, já que todas as pesquisas binárias são feitas em paralelo. Uma vez que um processador associado com um item x em $A \cup B$ conhece a colocação i em A e a colocação j em B, então este processador está pronto para escrever x na colocação $i + j$ do arranjo resultante. Assim, podemos unir A e B em tempo $O(\log n)$ usando n processadores.

Uma vez que sabemos como unir rapidamente duas listas ordenadas em paralelo, podemos usar isso como uma sub-rotina de uma versão paralela do algoritmo *mergesort*. Mais especificamente, para ordenar um conjunto S, dividimos S em dois subconjuntos S_1 e S_2 de igual tamanho e recursivamente os ordenamos em paralelo. Uma vez que tenhamos S_1 e S_2 ordenados, podemos usar o algoritmo de junção paralelo para uni-los em uma única lista ordenada. O número total de processadores necessários é n, uma vez que atribuímos a cada subproblema um número de processadores igual a seu tamanho. Da mesma forma, o tempo de execução do algoritmo, $T(n)$, caracteriza-se pela equação de recorrência $T(n) = T(n/2) + b \log n$, para uma constante b, o que implica que $T(n)$ é $O(\log^2 n)$. Assim, temos o seguinte:

Teorema 14.8: *dado um conjunto S de n itens, podemos ordenar S em tempo* $O(\log^2 n)$ *usando n processadores no modelo CREW PRAM.*

Observamos que o algoritmo acima é um exemplo do padrão simples de divisão e conquista paralelo. No Exercício C-14.23, exploramos como unir dois arranjos ordenados em tempo $O(\log n)$ usando $O(n/\log n)$ processadores, o que leva a um algoritmo de ordenação paralela ótimo. Como já foi dito, na verdade podemos ordenar n itens em tempo $O(\log n)$ usando n processadores no modelo EREW PRAM com um algoritmo mais sofisticado. Apesar de não descrevermos esse algoritmo, saber esse fato é útil para a construção de outros algoritmos paralelos eficientes. Daremos um exemplo desse tipo de aplicação na próxima subseção, aplicando junção e ordenação (e algumas das outras técnicas já apresentadas) a um problema geométrico.

14.2.6 Determinação do diâmetro de um polígono convexo

Consideremos o seguinte problema: dado um polígono convexo P, procure entre os pontos de P o par com maior distância entre si. Isso corresponde ao ***diâmetro*** de P, isto é, a distância entre o par de pontos de P mais afastados entre si. É fácil perceber que, para qualquer P, existe um par de pontos mais afastado que são ambos vértices de P. Assim, podemos facilmente resolver esse problema em tempo $O(\log n)$ usando $O(n^2)$ processadores (com a técnica simples de divisão e conquista para calcular a distância máxima entre todos $O(n^2)$ pares de vértices). Podemos fazer melhor do que isso, entretanto, tirando vantagem de estruturas geométricas apresentadas no problema e usando algumas das técnicas já analisadas. Nesta seção, mostramos o algoritmo DIAMETER para resolver o problema de determinação do diâmetro em tempo $O(\log n)$ usando $O(n \log n)$ processadores, o que é ótimo.

Notamos que os pontos p e q do par com maior distância entre si devem ser vértices ***antípodas*** de P. Os pontos p e q são antípodas se existirem duas linhas paralelas, L_1 e L_2, tangentes ao polígono P de maneira que L_1 contenha p e L_2 contenha q. Isso certamente reduz o número de pares que devem ser considerados de $O(n^2)$ para $O(n)$, mas ainda é um problema saber como enumerar os antípodas de forma eficiente em paralelo. A ordem cíclica dos vértices ao redor de P determina a direção de cada aresta.

Tratando cada aresta como um vetor, transladamos o conjunto de arestas-vetores para a origem. Qualquer linha através da origem desse diagrama de vetores intersecciona dois setores, o que corresponde a vértices antípodas (veja a Figura 14.9).

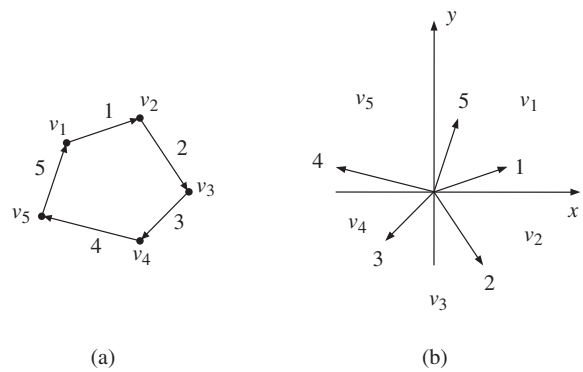


Figura 14.9 Tratando as arestas como vetores, transladamos essas arestas para a origem. Observamos que os vértices do polígono (a) correspondem a setores no diagrama de vetores (b).

Como queremos encontrar todos os vértices antípodas em tempo $O(\log n)$, não podemos usar o método de rotar uma linha contendo a origem através do conjunto de vetores. Também não podemos atribuir um processador a cada região (correspondente a um vértice v) e então enumerar todos os outros vértices que são antípodas a v por pesquisas binárias, pois pode haver $O(n)$ destes vértices para um v, e, além disso, temos apenas $O(n/\log n)$ processadores à disposição. Em vez disso, dividimos o conjunto de vetores em dois conjuntos, divididos pelo eixo x, rotamos todos os vetores abaixo do eixo x em 180° e usamos um procedimento de junção paralela para enumerar todos os vértices antípodas. Então, encontrando um máximo sobre os $O(n)$ pares, localizamos o par de pontos mais distantes de P (veja o Algoritmo 14.10).

Algoritmo DIAMETER(P):

Entrada: um polígono convexo $P = (v_1, v_2, ..., v_n)$.

Saída: dois pontos distantes, p e q, de P.

1. Usando a ordenação cíclica dos vértices de P para determinar a direção de cada aresta e tratando as arestas como vetores, translade o conjunto de arestas-vetores para a origem.
2. Rotule os vetores no diagrama de vetores que se localizem acima do eixo x de "vermelho" e os vetores abaixo do eixo x de "azul".
3. Rote cada vetor azul em um ângulo de 180°.
   {Os vetores vermelhos e azuis estão agora ordenados por ângulo}.
4. Use um algoritmo rápido de junção paralela para unir as duas listas ordenadas.
5. **para** cada vetor v **faça**

   Use a lista ordenada para calcular os dois vetores de cores opostas que antecedem e sucedem v na lista unida.

   Identifique cada par (a,b) como um par antípoda.
6. Encontre a maior das distâncias entre os pares antípodas para encontrar o par mais distante de P.

Algortimo 14.10 Algoritmo para encontrar os dois pontos mais distantes em um polígono convexo P.

Teorema 14.9: *o algoritmo DIAMETER encontra corretamente o par de pontos mais distantes em um polígono convexo P de n pontos em tempo* $O(\log n)$ *usando* $O(n/\log n)$ *processadores no modelo CREW PRAM.*

Prova: a correção do algoritmo DIAMETER vem do fato de que, ao girarmos os vetores ao redor do eixo x em 180°, todos os setores que eram opostos (isto é, ambos eram interseccionados pela mesma linha através da origem) agora estão sobrepostos. Voltamos aos limites da complexidade. Observamos que os Passos 1, 2, 3 e 5 podem ser resolvidos em tempo $O(1)$ se usarmos $O(n)$ processadores, uma vez que estamos executando trabalho $O(1)$ para cada um dos $O(n)$ objetos em cada um desses passos. Assim, podemos executar cada um desses passos em tempo $O(\log n)$ usando $O(n/\log n)$ processadores. Observamos também que os Passos 4 e 6 podem ser executados em tempo $O(\log n)$ usando $O(n/\log n)$ processadores (veja o Exercício C-14.23). ■

14.3 Algoritmos *on-line*

Um ***algoritmo on-line*** responde a uma seqüência de ***pedidos de serviço***, cada um associado a um ***custo***. Por exemplo, uma política de substituição de páginas da Web mantém páginas em uma memória *cache* sujeita a uma seqüência de pedidos de acesso, com o custo de um pedido de página sendo 0 se a página estiver na *cache*, e 1 se a página estiver fora da *cache*. Em um cenário ***on-line***, o algoritmo deve responder completamente a um pedido de serviço antes de receber o pedido seguinte. Se um algoritmo receber antecipadamente a lista de todos os pedidos que serão feitos, ele é chamado de algoritmo ***off-line***. Para analisar um algoritmo *on-line*, freqüentemente usamos uma análise competitiva, em que comparamos um algoritmo *on-line A* a um algoritmo

ótimo *off-line OPT*. Dada uma seqüência particular $P = (p_1, p_2,..., p_n)$ de pedidos de serviço, seja *cost* (A, P) o custo de A para P e seja *cost* (OPT, P) o custo do algoritmo ótimo *OPT* para P. O algoritmo A é chamado de ***c-competitivo*** para P se

$$cost(A, P) \leq c \cdot cost(OPT, P) + b,$$

para alguma constante $b \geq 0$. Se A for c-competitivo para toda seqüência P, então simplesmente dizemos que A é c-competitivo e chamamos c de ***razão de competitividade*** de A. Se $b = 0$, então dizemos que A tem uma razão de competitividade ***estrita*** com valor c.

O dilema do locatário

Um problema *on-line* bem conhecido, que pode ser explicado com uma pequena história, é chamado de ***dilema do locatário***: Alice decidiu usar um serviço de aluguel de músicas digitais para escutar algumas músicas de seu conjunto preferido, *Os Miadores*. Cada vez que Alice faz *download* de uma música, ela paga x unidades para "alugar" a música e ouvi-la, e o programa que ela usa obriga que ela pague novamente para tocar outra vez. Suponha que custe y unidades comprar todas as músicas do novo álbum de *Os Miadores*. Suponhamos, por exemplo, que y é 10 vezes maior do que x, ou seja, $y = 10x$. O dilema de Alice é decidir se e quando ela deve comprar o álbum completo em vez de acessar uma música de cada vez. Por exemplo, se ela comprar antes de acessar qualquer música e depois decidir que não gosta de nenhuma, então ela terá gastado 10 vezes mais do que devia. Mas se ela acessa muitas músicas e nunca compra o álbum, então ela potencialmente gastará mais do que deveria. Ou seja, se ela acessar músicas n vezes, então a estratégia de sempre alugar fará com que ela gaste $n/10$ vezes mais unidades do que deveria. Isto é, uma estratégia de comprar o álbum imediatamente tem uma razão de competitividade de pior caso de 10, e a estratégia de sempre alugar tem uma razão de competitividade de pior caso de $n/10$. Nenhuma dessas escolhas é muito boa.

Felizmente Alice tem uma estratégia com uma razão de competitividade 2: ela pode alugar por 10 vezes e então comprar o álbum. O cenário de pior caso é quando ela nunca escuta o álbum que acaba de comprar, e então ela terá gasto $10x + y = 2y$ unidades, quando deveria ter gasto y. Portanto, essa estratégia tem razão de competitividade 2. De fato, não importa quantas vezes y seja maior do que x, pois se Alice sempre alugar y/x vezes e então comprar, ela sempre terá uma razão de competitividade de 2.

14.3.1 Algoritmos de *cache*

Existem várias aplicações na Web que devem lidar com a informação apresentada em páginas da Web e acessá-las várias vezes. Esses acessos apresentam comportamento localizado, tanto no tempo quanto no espaço. Para explorar esse comportamento, é freqüentemente vantajoso armazenar cópias de páginas da Web em uma memória ***cache*** para que essas páginas possam ser recuperadas mais rapidamente quando acessadas novamente. Suponha que temos uma memória *cache* com m posições para conter páginas. Assumimos que uma página pode ser colocada em qualquer posição da *cache*. Isso é conhecido como uma memória *cache* ***completamente associativa***.

Quando um navegador trabalha, ele requisita diferentes páginas, e cada vez que uma página l é solicitada, o navegador determina (usando um teste rápido) se l não foi alterada e se está armazenada na *cache*. Se l estiver contida na *cache*, então o navegador preenche a solicitação usando a cópia. Se l não estiver na *cache*, então l é solicitada, transmitida pela Internet e transferida para a *cache*. Se uma das m posições estiver livre, então o navegador coloca l na posição livre. Se todas as m posições estiverem

ocupadas, então deve-se determinar qual a página que deve ser retirada para dar espaço à nova página *l*. Existem várias políticas diferentes que podem ser usadas para escolher esta página, e algumas das mais conhecidas políticas de substituição incluem as seguintes (veja a Figura 14.11):

- ***First-in, first-out* (FIFO)**: elimina a página que está na *cache* há mais tempo, ou seja, a página que foi transferida para a *cache* há mais tempo.
- **Página menos usada (LRU*)**: elimina a página que está na *cache* há mais tempo sem ser usada.

Além disso, podemos considerar uma estratégia simples e completamente aleatória:

- **Random**: elimina uma página qualquer dentre as que estão na *cache*.

A estratégia Random é de implementação fácil, pois ela apenas requer um mecanismo de geração de números aleatórios. O custo envolvido para implementar essa política é um trabalho $O(1)$ para cada página substituída. Além disso, não existe custo adicional por solicitação de página, a não ser determinar se uma página está na *cache* ou não. Essa política não tenta tirar vantagem da localidade temporal ou espacial que a navegação exibe tipicamente.

A estratégia FIFO é simples de implementar e apenas requer uma fila Q para armazenar referências às páginas na *cache*. As páginas são colocadas em Q quando são referenciadas por um navegador e trazidas para a *cache*. Quando uma página deve ser retirada, simplesmente retira-se a primeira da fila Q. Assim, essa política também requer trabalho adicional $O(1)$ por página substituída; portanto, a política FIFO não tem custo adicional por solicitação de página. Além disso, ela tenta tirar vantagem da localidade temporal.

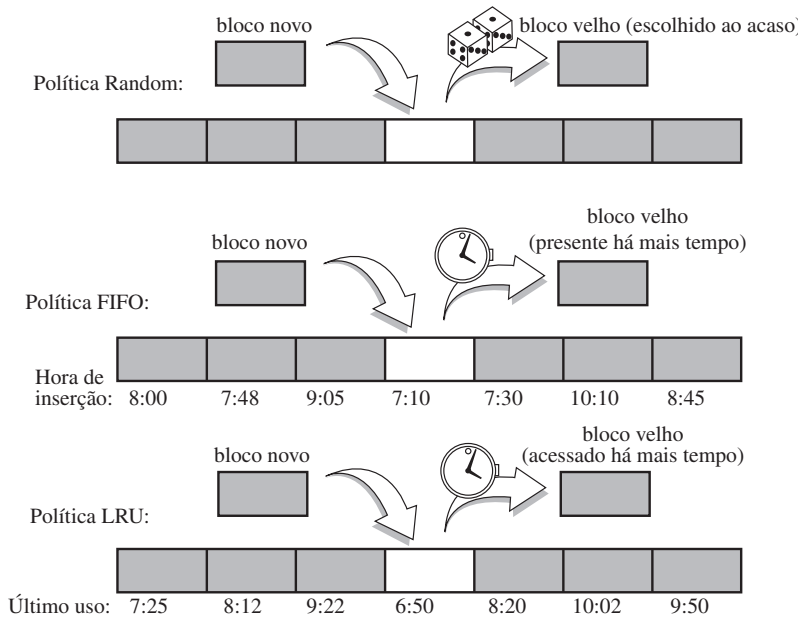


Figura 14.11 As políticas de substituição Random, FIFO e LRU.

* N. de R. Em inglês, *least recently used*.

A estratégia LRU dá um passo a mais do que a estratégia FIFO, que assume que a página que está na *cache* há mais tempo é a de requisição menos provável no futuro. A estratégia LRU tira vantagem explícita da localidade temporal maior possível, sempre descartando a página que foi usada há mais tempo. Do ponto de vista de decisão, esta é uma abordagem excelente, mas é custosa do ponto de vista de implementação: implementar a política LRU requer uma fila de prioridade Q que suporte a pesquisa por páginas existentes, por exemplo, usando apontadores ou algum tipo de localizador. Se Q for implementada como uma seqüência ordenada baseada em uma lista encadeada, então o custo para cada pedido de página e substituição é $O(1)$. Quando inserimos uma página em Q ou atualizamos sua chave, esta página recebe a maior chave em Q e é colocada ao final da lista, o que também pode ser feito em tempo $O(1)$. Mesmo que a estratégia LRU tenha um custo de tempo constante usando a implementação acima, os fatores constantes envolvidos (em termos de custo adicional de tempo e espaço extra para a fila de prioridade Q) tornam essa política menos atraente.

Como essas políticas diferentes de substituição têm diferentes características com relação à dificuldade de implementação e ao grau com que tiram vantagem da localidade, é natural solicitar uma análise comparativa entre esses métodos para determinar qual deles é o melhor.

Uma análise competitiva de pior caso para FIFO e LRU

Do ponto de vista do pior caso, as estratégias FIFO e LRU têm um comportamento competitivo pouco atraente. Por exemplo, suponha que temos uma *cache* contendo m páginas, e considere os métodos FIFO e LRU realizando a substituição de páginas para um programa com um laço que solicite repetidamente $m - 1$ páginas ciclicamente. Tanto a estratégia FIFO como a LRU não desempenham bem nesta situação, pois realizam uma substituição de página a cada pedido de página. Assim, do ponto de vista do pior caso, as duas estratégias são quase o pior que se pode imaginar — elas requerem uma substituição de página a cada requisição de página.

Esta análise de pior caso é um pouco pessimista, pois enfoca o comportamento de cada protocolo para uma seqüência ruim de requisições de páginas. Uma análise melhor seria feita comparando-se esses métodos entre todos os tipos possíveis de solicitações de páginas; porém, isso não pode ser feito de forma exaustiva. No entanto, já foram feitas várias simulações do comportamento dessas estratégias obtidas com programas reais. Com base nessas comparações experimentais, as políticas podem ser ordenadas de melhor para pior segundo esta ordem: (1) LRU, (2) FIFO e (3) Random. De fato, LRU é significativamente melhor que as outras para seqüências típicas de requisição de páginas, mas ainda tem baixo *desempenho* no pior caso, como o teorema a seguir demonstra.

Teorema 14.10: *as políticas de substituição de páginas LRU e FIFO têm razão de competitividade pelo menos m para uma cache de tamanho m.*

Prova: observamos anteriormente que existe uma seqüência $P = (p_1, p_2, ..., p_n)$ de requisições de páginas que faz com que FIFO e LRU realizem uma substituição a cada solicitação de página — o laço de $m + 1$ pedidos de página. Comparamos esse desempenho com aquele do algoritmo ótimo *off-line OPT*, que neste caso retira da *cache* a página que será solicitada daqui a mais tempo no futuro. Essa estratégia pode ser apenas implementada no caso *off-line*, quando temos toda a seqüência P fornecida antecipadamente. Quando aplicada ao laço de solicitações, a política *OPT* vai realizara uma substituição de página a cada m pedidos de página (pois ela retira da *cache* a página referenciada há menos tempo, já que esta será solicitada daqui a mais tempo no futuro). Assim, tanto FIFO quanto LRU são c-competitivas na seqüência P, onde

$$c = \frac{n}{n/m} = m.$$

Observe que, se qualquer porção $P' = (p_i, p_{i+1}, ..., p_j)$ de P fizer solicitações por m páginas diferentes (com p_{i-1} e/ou p_{j+1} não estando entre elas), então mesmo o algoritmo ótimo deve retirar uma página. Além disso, o maior número de páginas que FIFO e LRU retiram para uma porção P' deste tipo é m, a cada vez retirando uma página que fora referenciada antes de p_i. Portanto, FIFO e LRU têm uma razão de competitividade m, e esta é a melhor razão de competitividade possível para essas estratégias no pior caso. ■

O algoritmo com marcadores aleatórios

Mesmo que as políticas determinísticas FIFO e LRU tenham uma razão de competitividade de pior caso que não seja boa se comparada com um algoritmo *off-line* ideal, podemos mostrar que uma política randomizada que tenta simular LRU tem uma boa razão de competitividade. Especificamente, vamos estudar a razão de competitividade de uma estratégia randomizada que tenta emular a estratégia LRU, conhecida como estratégia de ***marcadores aleatórios***. Essa estratégia imita os melhores aspectos da política determinística LRU e, pela randomização, evita as situações de pior caso que prejudicam a estratégia LRU. A política de marcadores é a seguinte:

- **Marcador**: associa-se a cada página na *cache* uma variável booleana "marcado", que é inicializada com "false" para cada página na *cache*. Se um navegador requisitar uma página que está na *cache*, então a variável desta página é marcada como "true". Senão, se um navegador requisita uma página que não está na *cache*, então uma página aleatória cuja variável "marcado" esteja com o valor "false" é retirada da *cache* e substituída pela nova página, cuja variável "marcado" é imediatamente colocada em "true". Se todas as páginas da *cache* tiverem as variáveis marcadas como "true", então todas elas são remarcadas como "false" (veja a Figura 14.12).

Análise de competitividade de um algoritmo randomizado *on-line*

Com a definição acima, podemos realizar uma análise de competitividade da estratégia de marcadores. Antes dela, no entanto, precisamos definir o que chamamos de razão de competitividade de um algoritmo randomizado *on-line*. Uma vez que um algoritmo randomizado A como a estratégia de marcadores pode ter muitas execuções diferentes, de acordo com as escolhas aleatórias feitas, então definimos esse algoritmo como ***c-competitivo*** para uma seqüência de pedidos P, se

$$E(cost(A, P)) \leq c \cdot cost(OPT, P) + b$$

para alguma constante $b \geq 0$, onde $E(cost(A, P))$ denota o custo esperado do algoritmo A na seqüência P (com esse custo esperado sendo tomado sobre todas as escolhas do algoritmo A). Se A for ***c-competitivo*** para toda seqüência P, então dizemos que A é ***c-competitivo***, e chamamos c de ***razão de competitividade*** de A.

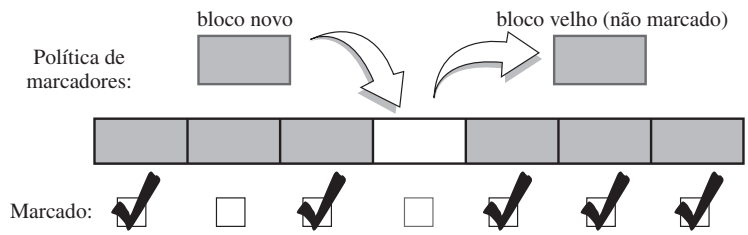


Figura 14.12 Estratégia de substituição com marcadores.

Teorema 14.11: *a política de substituição de páginas com marcadores para uma cache com m páginas tem razão de competitividade* $2 \log(m)$.

Prova: seja $P = (p_1, p_2, ..., p_n)$ uma seqüência suficientemente longa de pedidos de páginas. A política de marcadores particiona implicitamente esses pedidos em ***rodadas***. Cada rodada começa com todas as páginas na *cache* tendo o marcador em "false", e uma rodada termina quando todas as páginas na *cache* tiverem o marcador em "true" (com o próximo pedido iniciando a rodada seguinte, pois a política então altera todos os marcadores para "false" novamente). Considere a i-ésima rodada em P, e chame uma página requisitada na rodada i de ***nova*** se ela não estiver na *cache* quando a rodada i se inicia. Também chamamos uma página que está na *cache* de ***pendente*** se ela tiver o marcador em "false". Assim, no início da rodada i, todas as páginas na *cache* são pendentes. Seja m_i o número de páginas novas referenciadas na rodada i, e seja b_i o número de páginas na *cache* para o algoritmo *OPT* no início da rodada i e que não estão na *cache* usando marcadores naquele momento. Como a política de marcadores tem de fazer uma substituição de página a cada um dos m_i pedidos, o algoritmo *OPT* deve realizar pelo menos $m_i - b_i$ substituições de página na rodada i (veja a Figura 14.13). Além disso, como cada uma das páginas na *cache* de marcadores ao final da rodada i foi so-

licitada durante a mesma rodada, o algoritmo *OPT* deve realizar pelo menos $m_i - b_i$ substituições de página na rodada i. Assim, o algoritmo *OPT* deve realizar pelo menos

$$\text{máx}\{m_i - b_i, b_{i+1}\} \geq \frac{m_i - b_i + b_{i+1}}{2}$$

substituições de página na rodada i. Somando para todas as k rodadas definidas por P, vemos que o algoritmo *OPT* deve realizar pelo menos o seguinte número de substituições de páginas:

$$L = \sum_{i=1}^{k} \frac{m_i - b_i + b_{i+1}}{2} = (b_{k+1} - b_1)/2 + \frac{1}{2}\sum_{i=1}^{k} m_i.$$

A seguir vamos considerar o número esperado de substituições de página realizado pela política de marcadores.

Já observamos que a política de marcadores tem de realizar pelo menos m_i substituições de página na rodada i. Ela pode realizar mais do que isso, no entanto, se ela descartar páginas pendentes que serão solicitadas mais tarde na mesma rodada. Assim, o número esperado de substituições de página na política de marcadores é $m_i + n_i$, onde n_i é o número esperado de páginas pendentes que são solicitadas na rodada i depois de serem retiradas da *cache* na mesma rodada. O valor n_i é igual à soma, para todas as páginas pendentes referenciadas na rodada i, da probabilidade de que essas páginas estejam fora da *cache* quando forem referenciadas. Quando uma página pendente v é referenciada na rodada i, a probabilidade de que v esteja fora da *cache* é no máximo f/g, onde f é o número de páginas novas referenciadas antes de v, e g é o número de páginas pendentes que ainda não foram referenciadas. Isso acontece porque cada referência a uma página nova retira da *cache* uma página pendente e marcada "false". O custo da política de marcadores vai ser máximo quando todos os m_i pedidos de páginas novas forem feitos antes de todos os pedidos por páginas pendentes. Assumindo este pior caso, o número esperado de páginas pendentes retiradas da *cache* e posteriormente referenciadas na rodada i pode ser limitado pela expressão

$$\begin{aligned} n_i &\leq \frac{m_i}{m} + \frac{m_i}{m-1} + \frac{m_i}{m-2} + \cdots + \frac{m_i}{m_i+1} \\ &\leq m_i \sum_{j=1}^{m} \frac{1}{j}, \end{aligned}$$

pois existem $m - m_i$ referências a páginas pendentes na rodada i. Essa soma é conhecida como o m-ésimo número harmônico, e denotada por H_m. Então, temos

$$n_i \leq m_i H_m.$$

Assim, o número esperado de substituições de páginas usando a política de marcadores é no máximo

$$U = \sum_{i=1}^{k} m_i (H_m + 1) = (H_m + 1)\sum_{i=1}^{k} m_i.$$

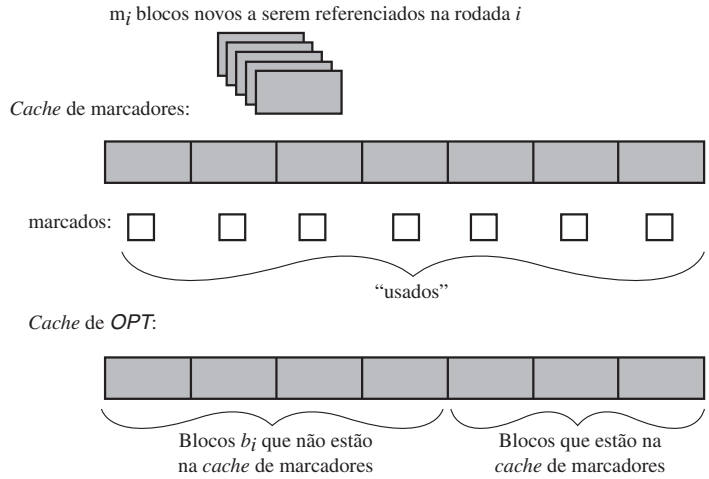


Figura 14.13 O estado da *cache* de marcadores e a *cache* de *OPT* no início da rodada i.

Portanto, a razão de competitividade da política de marcadores é no máximo

$$\begin{aligned} \frac{U}{L} &= \frac{(H_m+1)\sum_{i=1}^{k} m_i}{(1/2)\sum_{i=1}^{k} m_i} \\ &= 2(H_m+1). \end{aligned}$$

Usando uma aproximação para H_m, a saber, $H_m \leq \log m$, a razão de competitividade para a política de marcadores é no máximo $2\log(m)$. ■

Assim, a análise de competitividade mostra que a política de marcadores é razoavelmente eficiente.

14.3.2 Estratégias de leilão

Nesta seção, mostraremos como aplicar a análise de competitividade a alguns problemas simples envolvendo leilões na Web. O cenário que temos é de um objeto como uma antigüidade, como uma obra de arte ou uma jóia, que tem um valor inteiro entre 1 e B unidades, e desejamos vendê-lo em um leilão. Neste leilão, devemos fornecer um algoritmo A que recebe um lance de cada vez e aceita ou rejeita o lance antes de conhecer o seguinte (se houver um seguinte, é claro). A meta é maximizar o total que A obtém pelo objeto.

Análise de competitividade para um problema de maximização

Como desejamos maximizar um lucro em vez de minimizar um custo, precisamos redefinir o que queremos dizer com um algoritmo c-competitivo. Diremos que um algoritmo é c-competitivo se

$$cost(A, P) \geq cost(OPT, P)/c + b,$$

para alguma constante $b \geq 0$. Assim como no caso de minimização, se $b = 0$, então dizemos que o algoritmo de maximização é estritamente c-competitivo.

Se não sabemos quantos lances teremos, no entanto, então um algoritmo determinístico não tem escolha além de aceitar o primeiro lance, pois ele pode ser o único. Pela análise de competitividade, se compararmos esta estratégia com a de um adversário que sabe que lances serão feitos, então teremos uma razão de competitividade valendo B. Essa baixa razão de competitividade se deve ao fato de que $P_1 = (1, b)$ e $P_2 = (1)$ são seqüências de lances válidas. Como não sabemos se lidamos com P_1 ou P_2, então temos de aceitar o lance de uma unidade assim que o recebemos, enquanto um adversário que sabe que a seqüência é P_1 vai esperar o lance de B unidades. Se soubermos ao menos o número n de lances a esperar, podemos fazer melhor do que isso.

Teorema 14.12: *se o número de lances n for conhecido, então existe um algoritmo determinístico para o problema do leilão com razão de competitividade* $O\left(\sqrt{B}\right)$.

Prova: o algoritmo que obtém essa razão de competitividade é bastante simples: ele aceita o primeiro lance acima de $\lfloor \sqrt{B} \rfloor$. Se não houver tal lance, então aceita o último lance. Para analisar a razão de competitividade, existem dois casos:

1. Suponha que não haja lance acima de $\lfloor \sqrt{B} \rfloor$, e seja m o maior lance dado. Então, no pior caso, poderemos aceitar um lance de uma unidade enquanto o algoritmo *off-line* aceita o lance m. Assim, temos uma razão de competitividade de $m \leq \lfloor \sqrt{B} \rfloor$.
2. Suponha que haja um lance acima de $\lfloor \sqrt{B} \rfloor$, e seja m o maior lance dado. Então, no pior caso, podemos aceitar um lance de $\lfloor \sqrt{B} \rfloor + 1$ unidades enquanto o algoritmo *off-line* aceita um lance de valor m. Assim, obtemos uma razão de competitividade de $m/(\lfloor \sqrt{B} \rfloor + 1)$, e este valor é certamente menor do que $\sqrt{B}$. ■

Essa razão de competitividade não é muito boa, mas é a melhor que se pode obter com um algoritmo determinístico.

Teorema 14.13: *se o número de lances n for conhecido, então nenhum algoritmo determinístico para o problema do leilão pode ter uma razão de competitividade melhor do que* $\Omega(\sqrt{B})$.

Prova: seja A um algoritmo determinístico para o problema do leilão, e seja $b = \lfloor \sqrt{B} \rfloor$. Considere o comportamento de A para a seqüência $P = (b, B)$, que é dada *on-line*. Há dois casos:

1. Se A aceitar b, então o adversário aceita o lance B, e a razão de competitividade é $\Omega(\sqrt{B})$ neste caso.
2. Se A não aceitar o lance de valor b, como A é *on-line* no momento da rejeição, não é possível saber se a seqüência recebida é P ou $Q = (b, 1)$ em seu lugar. Assim, podemos considerar o comportamento de A para Q: como A não conhece o próximo lance quando precisa decidir sobre b, então A é forçado a aceitar o primeiro lance. Neste caso, a razão de competitividade de A tem valor b, que é $\Omega(\sqrt{B})$. ■

Assim, um algoritmo determinístico não pode obter uma razão de competitividade muito boa para o problema do leilão.

Limitação aleatória

Vamos considerar o uso de um algoritmo randomizado para o problema do leilão. Neste caso, podemos obter uma razão de competitividade bem melhor mesmo que o algoritmo não saiba antecipadamente o número de lances a serem feitos. Isso é conseguido com uma técnica de ***limitação aleatória***, descrita a seguir.

O algoritmo A escolhe um inteiro i aleatoriamente no conjunto $\{0, 1, 2, ..., \log B\}$ e aceita o primeiro lance que seja maior ou igual a 2^i. Como o teorema a seguir demonstra, esse algoritmo tem uma razão de competitividade de $O(\log(B))$.

Teorema 14.14: *o algoritmo de limitação aleatória tem uma razão de competitividade que é* $O(\log(B))$.

Prova: seja P a seqüência de lances e m o maior lance em P. Então, o algoritmo *off-line* aceita o valor de m. Lembre que a razão de competitividade de um algoritmo randomizado é baseada no custo esperado, que no caso do problema do leilão deveria ser visto como "benefício", pois estamos tentando maximizar o lance aceito. O valor esperado do lance aceito por A é pelo menos $m/(1 + \log B)$, pois A irá aceitar m com probabilidade $1/(1 + \log B)$. Portanto, a razão de competitividade para A é no máximo $1 + \log B$. ■

Na próxima subseção, mostraremos de que forma a análise competitiva pode ser aplicada ao problema do projeto de estruturas de dados.

14.3.3 Árvores de pesquisa competitivas

Nesta subseção, apresentamos uma estrutura de dicionário baseada em árvore adaptativa, que é balanceada e competitiva. Nossa abordagem é baseada em um parâmetro de ***energia potencial*** armazenado em cada nodo da árvore. Quando consultas e atualizações são feitas, a energia potencial de um nodo é aumentada ou diminuída. Quando a energia potencial de um nodo atinge um determinado limite, reconstruímos a árvore que tem aquele nodo como raiz. Mostramos que, apesar da simplicidade, esse esquema é competitivo.

Árvores binárias de pesquisa com balanceamento de energia

Lembre que um ***dicionário*** guarda pares de chaves ordenadas e elementos, sujeitos a atualizações e consultas. Uma forma comum de implementar o TAD dicionário é usar uma árvore binária balanceada de pesquisa que mantenha o equilíbrio por rotações locais. Essas rotações são tipicamente rápidas, mas, se a árvore tiver estruturas auxiliares, então as rotações são freqüentemente lentas. Descrevemos uma estrutura simples para uma árvore que mantém equilíbrio sem rotações usando um parâmetro de energia potencial armazenado em cada nodo e com reconstrução parcial. Nossa abordagem não usa regras explícitas de balanceamento e, em vez disso, usa rótulos para os potenciais nos nodos, permitindo uma estrutura simples, adaptativa e competitiva com outras abordagens.

Uma ***árvore com balanceamento de energia*** é uma árvore binária de pesquisa T em que cada nodo v armazena um elemento e, tal que todos os elementos na subárvore esquerda de v são menores ou iguais a e, e todos os elementos na subárvore direita de v são maiores ou iguais a e. Cada nodo v em T mantém um parâmetro n_v, que é o número de elementos armazenados na subárvore cuja raiz é v, incluindo v. Cada nodo v

em T também mantém um parâmetro de energia potencial p_v. Inserções e remoções são tratadas como em uma árvore binária comum, não-balanceada, com uma pequena modificação: cada vez que realizamos uma operação de atualização que percorre um caminho da raiz de T até um nodo w em T, incrementamos p_v em 1 para cada nodo v no caminho. Se não existir um nodo v neste caminho para o qual $p_v \geq n_v/2$, então terminamos. Senão, seja v o nodo mais alto em T tal que $p_v \geq n_v/2$. Reconstruímos a subárvore com raiz v como uma árvore binária completa, e zeramos os campos de potencial de cada nodo nesta subárvore (incluindo o próprio v). Esse é o algoritmo completo para realização de atualizações.

Essa estratégia simples imediatamente implica o seguinte.

Teorema 14.15: *o pior caso para a altura de uma árvore com balanceamento de energia é* $O(\log(n))$, *e o tempo amortizado para realizar atualizações nesta árvore é* $O(\log(n))$.

Prova: é suficiente mostrar que $n_w < n_v/4$ para qualquer nodo v com irmão w. Vamos supor que isso não seja verdade. Então, desde o último balanceamento feito no pai z de v e w (quando os tamanhos das subárvores de v e w eram iguais), o número de remoções na subárvore de w mais o número de inserções na subárvore de v deve ter sido pelo menos $3n_v/4$. Ou seja, $p_z \geq 3n_v$. Neste momento, $n_z = n_w + n_v < 5n_v/4$. Portanto, $p_z \geq 3n_v/4$. No entanto, isso não pode acontecer, pois teríamos reconstruído a subárvore de z assim que tivéssemos $p_z > n_z/2$.

O fato acima implica que a altura de uma árvore binária T seja $O(\log(n))$, portanto o tempo de pior caso para qualquer procura é $O(\log(n))$. Além disso, cada vez que uma árvore é reestruturada a partir de uma raiz v, temos que $p_v \geq A_v/4$. Ou seja, p_v é $O(n_v)$. Como custa tempo $O(n_v)$ reestruturar a árvore de v, isso implica que podemos cobrar as p_v operações anteriores que depositam "energia" no nodo v para esta reestruturação. Como a profundidade da árvore com balanceamento de energia é $O(\log(n))$ e qualquer operação de atualização vai depositar energia em um caminho da raiz a uma folha, isso implica que cada operação de atualização vai ser cobrada no máximo $O(\log(n))$ vezes pela reestruturação. Assim, o tempo amortizado necessário para realizar operações de atualização em uma árvore de pesquisa com balanceamento de energia é $O(\log(n))$. ■

Árvores de pesquisa com balanceamento de energia e distribuições não-homogêneas

Nossa abordagem baseada em energia potencial pode ser estendida para adaptar-se a um dicionário com distribuições não-homogêneas nas procuras e atualizações. Aumentamos a árvore T neste caso de forma que cada nodo v armazene um ***contador de acessos*** a_v que conta o número de vezes que o elemento armazenado em v foi acessado. Cada vez que um nodo é acessado em uma pesquisa, incrementamos seu contador de acessos e também incrementamos o parâmetro de energia potencial de cada nodo no caminho de v até a raiz. Mantemos o mesmo algoritmo de inserção, mas agora, quando removemos um nodo v, incrementamos a energia potencial de cada nodo no caminho de v até a raiz em a_v. Seja A_v o valor acumulado dos contadores de acesso em todos os nodos da subárvore com raiz v em T. Realizamos um passo de reconstrução cada vez que a energia potencial de um nodo aumenta mais do que um quarto do seu contador de acessos acumulado, ou seja, sempre que $p_v \geq A_v/4$. Nesta árvore binária adaptada, reconstruímos a subárvore de forma que os nodos sejam balanceados apro-

ximadamente por seus contadores de acessos, ou seja, tentamos balancear os filhos por seus valores de A_v. Especificamente, existe um algoritmo de tempo linear para a construção de uma árvore (veja o Exercício C-3.25) que garante que, para qualquer nodo v com pai z, $A_z \geq 3A_v/2$. Para qualquer nodo v diferente da raiz, usamos $\hat{A}_v$ para denotar o tamanho da subárvore com raiz v mais o peso do item armazenado no pai z de v (portanto, $A_z = A_v + \hat{A}_w$, onde w denota o irmão de v).

Lema 14.16: *para qualquer nodo v com irmão w*, $\hat{A}_w \geq A_v/8$.

Prova: para provar uma contradição, suponha que $\hat{A}_v < A_w/8$. Então, desde o último rebalanceamento no pai z de v e w (quando $\hat{A}'_w < A'_v/2$, onde $\hat{A}'_w$ e A'_v representam os valores anteriores de $\hat{A}_w$ e A_v respectivamente), o peso total das remoções na subárvore de w mais o número de inserções e acessos na subárvore de v mais os acessos que terminam no pai de v devem ter sido pelo menos $3A_v/8$. Ou seja, $p_z \geq 3A_v/8$. Neste momento, temos que $A_z = \hat{A}_w + A_v < 9A_v/8$ e, portanto, $p_z \geq 3A_v/8 > A_z/3$. Mas isto não pode acontecer, pois teríamos de reconstruir a subárvore de z quando $p_z > A_z/4$. ■

A implicação do fato acima é que um elemento tendo o contador de acessos a é armazenado em um nodo com profundidade $O(\log A/a)$ na árvore, onde A é a freqüência total de acessos dos nodos na árvore, como o teorema a seguir demonstra.

Teorema 14.17: *um elemento tendo freqüência de acessos a é armazenado em uma profundidade* $O(\log A/a)$*, onde A é a freqüência total de acessos dos nodos na árvore.*

Prova: pelo lema anterior, o contador total de acessos na subárvore com raiz v é maior (pelo menos em um fator constante) do que o total de acessos para os filhos de v. Assim, a profundidade de qualquer nodo v na árvore é no máximo $O(\log A/a_v)$. ■

Antes que analisemos este tipo de árvore, vamos estabelecer o lema a seguir. Ele é utilizado para analisar o tempo necessário para processar as operações de uma subseqüência S_v da seqüência completa S, formada por todas as operações que acessam ou atualizam o elemento em um nodo v.

Lema 14.18: *seja* A_i *o total dos contadores de acesso de todos os nodos presentes na árvore com balanceamento de energia (para S) depois de realizarmos a i-ésima operação em* S_v*. Então,*

$$\sum_{i=1}^{m} \log A_i / i$$

é

$$O(m \log \hat{A}/m),$$

onde $m = |S_v|$, *e* $\hat{A}$ *é o total dos contadores de acesso para todos os elementos referenciados em S.*

Prova: vamos assumir que m é uma potência de 2 para facilitar a análise, ou seja, $m = 2^k$ para algum k. Note que

$$\sum_{i=1}^{m} \log \frac{A_i}{i} \leq \sum_{i=1}^{m} \log \frac{\hat{A}}{i} = \sum_{i=1}^{m} \log \left(\frac{\hat{A}}{m}\right)\left(\frac{m}{i}\right)$$

$$= \sum_{i=1}^{m} \log \frac{\hat{A}}{m} + \sum_{i=1}^{m} \log \frac{m}{i} = m \log \frac{\hat{A}}{m} + \sum_{i=1}^{m} \log \frac{m}{i}.$$

Assim, para estabelecer o lema, precisamos apenas limitar o último termo acima (o somatório). Note que

$$\begin{aligned}\sum_{i=1}^{m}\log\frac{m}{i} &= \sum_{i=1}^{2^k}\log\frac{2^k}{i} \le \sum_{i=1}^{2^k}\log\frac{2^k}{2\lfloor\log i\rfloor} = \sum_{i=1}^{2^k} k-\lfloor\log i\rfloor \\ &\le \sum_{j=1}^{k} j2^{k-j} = 2^k\sum_{j=1}^{k}\frac{j}{2^j} \le 2\cdot 2^k = 2m.\end{aligned}$$

■

O lema anterior é necessário para relacionar os contadores de acesso vistos até agora em uma seqüência de atualizações e acessos aos contadores de acesso para toda a seqüência. Um oráculo, que chamaremos de ***oráculo da árvore***, conhecendo a seqüência antecipadamente, poderia construir uma árvore estática baseada nos contadores de acesso conhecidos, de forma que o tempo de execução de cada operação de acesso ou atualização em um nodo v seja $O(\log \hat{A}/\hat{a}_v)$, onde $\hat{a}_v$ é o contador de acessos para o elemento no nodo v.

Teorema 14.19: *a árvore de pesquisa com balanceamento de energia tem um desempenho amortizado para operações de atualização a cada nodo v de* $O(\log \hat{A}/\hat{a}_v)$, *que está a um fator constante do desempenho que pode ser atingido pela árvore baseada em oráculo.*

Prova: seja S uma seqüência de n operações de dicionário e seja T a árvore estática construída pela árvore baseada em oráculo. Considere uma subseqüência S_v de S formada por todas as operações que acessam ou atualizam os elementos em um dado nodo v. Seja A_i o total dos contadores de acesso dos nodos presentes na árvore com balanceamento de energia (para S) depois de realizarmos a i-ésima operação em S_v. Note que o tempo de execução amortizado para realizar a i-ésima operação em S_v usando a árvore com balanceamento de energia é proporcional à futura profundidade de v na árvore com balanceamento de energia, que será no máximo $O(\log A_i/i)$. Assim, o tempo amortizado exigido para realizar todas as operações em S_v é proporcional a no máximo

$$\sum_{i=1}^{m}\log A_i/i,$$

enquanto o tempo total requerido pela implementação da árvore baseada em oráculo é de

$$m\log \hat{A}/\hat{a}_v = m\log \hat{A}/m,$$

onde $m = |S_v|$. Entretanto, pelo Lema 14.18,

$$\sum_{i=1}^{m}\log A_i/i$$

é

$$O(m\log \hat{A}/m),$$

o que implica que o desempenho em termos de tempo da abordagem com balanceamento de energia para S_v é, no máximo, um fator constante maior do que para a árvore com oráculo. Portanto, uma afirmação similar é válida para todo o processamento de S. ■

Assim, apesar da simplicidade, a árvore com balanceamento de energia é eficiente e competitiva.

14.4 Exercícios

Reforço

R-14.1 Descreva em detalhes os algoritmos de inserção e remoção para uma árvore (a, b).

R-14.2 Suponha que T seja uma árvore genérica em que cada nodo interno tem pelo menos cinco e no máximo oito filhos. Quais valores de a e b fazem de T uma árvore (a, b) válida?

R-14.3 Para quais valores de d a árvore T do exercício anterior é uma árvore-B de ordem d?

R-14.4 Desenhe a árvore-B de ordem 7 que resulta da inserção das chaves a seguir (na ordem dada) em uma árvore vazia T:

(4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12).

R-14.5 Mostre cada nível de recursão ao se realizar um *merge-sort* com memória externa e de quatro vias para a seqüência dada no exercício anterior.

R- 14.6 Considere a generalização do dilema do locatário em que Alice pode comprar ou alugar esquis de neve separadamente das botas. Digamos que alugar os esquis custa a unidades, e comprar os esquis custa b unidades. De forma similar, alugar as botas custa c unidades, e comprá-las custa d unidades. Descreva um algoritmo *on-line* 2-competitivo para Alice tentar minimizar os custos de esquiar, mas assuma que ela não sabe ainda por quantas vezes irá esquiar no futuro.

R-14.7 Considere uma *cache* de memória inicialmente vazia e composta de quatro páginas. Quantas vezes a página desejada não estará na *cache* se usarmos o algoritmo LRU e a seqüência pedida for (2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)?

R-14.8 Considere uma *cache* de memória inicialmente vazia e composta de quatro páginas. Quantas vezes a página desejada não estará na *cache* se usarmos o algoritmo FIFO e a seqüência pedida for (2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)?

R-14.9 Considere uma *cache* de memória inicialmente vazia e composta de quatro páginas. Quantas vezes a página desejada não estará na *cache* se usarmos o algoritmo baseado em marcadores e a seqüência pedida for (2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)? Mostre as decisões randômicas de seu algoritmo.

R-14.10 Considere uma *cache* de memória inicialmente vazia e composta de quatro páginas. Construa uma seqüência de pedidos de páginas que faça o algoritmo baseado em marcadores passar por quatro rodadas.

R-14.11 Mostre como um algoritmo paralelo usando recursão dupla determina os prefixos paralelos da seqüência (1, 4, 20, 12, 7, 15, 32, 10, 9, 18, 11, 45, 22, 50, 5, 16).

Criatividade

C-14.1 Mostre como implementar um dicionário em memória externa usando uma seqüência não-ordenada de forma que as atualizações requeiram apenas $O(1)$ transferências e $O(n/B)$ transferências no pior caso, onde n é o número de elementos, e B é o número de nodos da lista que cabem em um bloco do disco.

C-14.2 Altere as regras que definem uma árvore vermelho-preta de forma que cada árvore vermelho-preta T tenha uma árvore (4, 8) correspondente e vice-versa.

C-14.3 Descreva uma versão modificada do algoritmo de inserção em árvore-B de forma que cada vez que criamos um *overflow* por causa da divisão de um nodo v, redistribuímos chaves entre todos os filhos de v de forma que cada filho tenha aproximadamente o mesmo número de chaves (possivelmente fazendo a divisão também no pai de v). Qual a menor fração de cada bloco que estará sempre preenchida usando este algoritmo?

C-14.4 Outra implementação possível de dicionário usando memória externa é usar uma *skip list*, mas para coletar grupos consecutivos de $O(B)$ nodos em blocos individuais em qualquer nível da *skip list*. Definimos uma ***B-skip list*** de ordem d como sendo uma representação desse tipo, em que cada bloco contém pelo menos $\lceil d/2 \rceil$ e no máximo d nodos da lista. Vamos escolher d como o máximo número de nodos da lista em um dado nível que seja capaz de caber em um bloco. Descreva como devemos modificar o algoritmo de inserção e remoção para uma *B-skip list* de forma que a altura esperada da estrutura seja $O(\log(n)/\log(B))$.

C-14.5 Suponha que, em vez de termos a função de procura de nodos $f(d) = 1$ em uma árvore-B de ordem d, temos $f(d) = \log(d)$. Como se comporta o tempo de execução assintótico para realizar uma procura nesta árvore?

C-14.6 Descreva como usar uma árvore B para implementar o TAD fila de forma que o número total de transferências de disco necessárias para processar uma seqüência de n operações de enqueue e dequeue seja $O(n/B)$.

C-14.7 Descreva como usar uma árvore-B para implementar o TAD partição (da Seção 4.2.2) de forma que as operações union e find usem no máximo $O(\log(n)/\log(B))$ transferências de disco.

C-14.8 Suponha que temos uma seqüência S de n elementos com chaves inteiras de forma que alguns elementos em S são "azuis" e outros são "vermelhos". Além disso, dizemos que um elemento vermelho e ***faz par*** com um elemento azul f se eles tiverem a mesma chave. Descreva um algoritmo eficiente de memória externa para encontrar todos os pares em S. De quantas transferências de disco seu algoritmo precisa?

C-14.9 Considere o problema de manutenção de páginas na *cache* em que a *cache* pode guardar m páginas e em que temos uma seqüência P de n pedidos que podem ser por qualquer uma das $m + 1$ páginas. Descreva uma estratégia ótima para o algoritmo *off-line* e mostre que ele causa no máximo $m + n/m$ falhas da *cache* ao ser iniciado com uma *cache* vazia.

C-14.10 Considere a estratégia de manutenção de páginas baseada no mecanismo de *página menos usada* (LFU) em que a página da *cache* que foi acessada menos freqüentemente é escolhida para deixar a *cache* quando uma nova página tiver de entrar. Se existem empates, a LFU descarta a página acessada menos freqüentemente e que está na cache há mais tempo. Mostre que existe uma seqüência P de n pedidos que faz com que a LFU "erre" $\Omega(n)$ vezes para uma *cache* de m páginas, enquanto o algoritmo ótimo erra apenas m vezes.

C-14.11 Demonstre que LRU é m-competitivo para qualquer seqüência de n pedidos de página, onde m é o tamanho da *cache*.

C-14.12 Demonstre que FIFO é m-competitivo para qualquer seqüência de n pedidos de página, onde m é o tamanho da *cache*.

C-14.13 Qual o número esperado de substituições de blocos realizadas pela política Random em uma *cache* de tamanho m para uma seqüência de acessos de tamanho n que acessa iterativamente $m + 1$ blocos de forma cíclica (assumindo que n é muito maior do que m)?

C-14.14 Demonstre que o algoritmo baseado em marcadores é H_m-competitivo quando o tamanho da *cache* for m e existirem $m + 1$ páginas que podem ser acessadas, onde m denota o m-ésimo número harmônico.

C-14.15 Demonstre como fazer a junção de dois arranjos de n elementos A e B com elementos distintos em tempo $O(\log(n))$ usando $O(n/\log(n))$ processadores no modelo CREW PRAM.

C-14.16 Demonstre como um EREW PRAM E com p processadores pode simular qualquer CREW PRAM C com p processadores, de forma que cada etapa paralela em C pode ser simulada em tempo $O(\log(p))$ em E.

C-14.17 Descreva um algoritmo paralelo para multiplicar duas matrizes $n \times n$ em tempo $O(\log^2(n))$ usando n^3 processadores.

C-14.18 Descreva um algoritmo paralelo para o modelo CRCW PRAM que determine o AND de n *bits* em tempo $O(1)$ usando n processadores.

C-14.19 Descreva um algoritmo paralelo para o modelo CRCW PRAM que determine o máximo de n números em tempo $O(1)$ usando n processadores.

C-14.20★ Descreva um algoritmo paralelo para o modelo CRCW PRAM que determine o máximo de n números em tempo $O(\log \log(n))$ usando $O(n)$ processadores.

Dica: use divisão e conquista paralela em que divide-se primeiro o conjunto de números em $\sqrt{n}$ grupos de tamanho $\sqrt{n}$ cada.

C-14.21 Descreva um algoritmo paralelo para o modelo CRCW PRAM capaz de ordenar uma seqüência S de n inteiros na faixa de valores $[1, c]$ para uma constante $c > 1$ em tempo $O(\log(n))$ usando $O(n/\log(n))$ processadores.

C-14.22 Descreva um algoritmo paralelo para o modelo CRCW PRAM que execute em tempo $O(\log(n))$, usando n processadores, a cobertura convexa de n pontos no plano (veja o Capítulo 12) que estejam armazenados em um arranjo e ordenados por suas coordenadas x (assuma que as coordenadas x são todas diferentes).

C-14.23 Sejam A e B dois arranjos ordenados com n inteiros cada. Descreva um método para unificar A e B em um único arranjo C em tempo $O(\log(n))$ usando $O(n/\log(n))$ processadores no modelo CREW PRAM.

Dica: comece associando um processador a cada $\lceil \log n \rceil$-ésima posição em A (respectivamente em B) e realizando uma pesquisa binária por este item no outro arranjo.

Projetos

P-14.1 Escreva uma classe que implemente todos os métodos de um dicionário dado na Seção 2.5 com uma árvore (a, b), onde a e b são constantes inteiras.

P-14.2 Implemente uma árvore B assumindo que o bloco de disco seja 1.000 e as *chaves* sejam inteiras. Teste o número de transferências do disco necessários para processar uma seqüência de operações do dicionário.

Notas

Knuth [119] contém boas discussões sobre ordenação e pesquisa em memória externa, e Ullman [203] discute estruturas de memória externa para bancos de dados. O leitor interessado no estudo da arquitetura de sistemas de memória hierárquica deve consultar o capítulo de Burger *et al.* [43] ou o livro de Hennessy e Patterson [93]. O livro de Gonnet e Baeza-Yates [81] compara o desempenho de vários algoritmos de ordenação, muitos dos quais de memória externa. Árvores-B foram inventadas por Bayer e McCreight [24], e Comer [51] fornece uma boa descrição dessa estrutura de dados. Os livros de Mehlhorn [148] e Samet [176] também têm boas discussões sobre árvores-B e suas variações. Aggarwal e Vitter [5] estudam a complexidade de E/S da ordenação e outros problemas relacionados estabelecendo limites superiores e inferiores, o que inclui o limite inferior para ordenação dado neste capítulo. Goodrich *et al.* [87] estudam a complexidade de E/S de vários problemas de geometria computacional. O leitor interessado em um estudo maior de algoritmos com E/S eficientes é encorajado a examinar o artigo de Vitter [206].

Para uma boa introdução geral ao projeto de algoritmos paralelos, aconselham-se os livros de JáJá [107], Reif [172], Leighton [130] e Akl e Lyons [11]. Para a discussão de desenvolvimentos mais recentes no projeto de algoritmos paralelos, veja os capítulos de Atallah e Chen [16] e Goodrich [84]. Brent [39] apresentou o padrão para o que chamamos de "teorema de Brent" em 1976. Kruskal *et al.* [126] estudam o problema do prefixo paralelo e suas aplicações, e Cole e Vishkin [50] estudam este e o problema de colocação em detalhes. Em [186] Shiloach e Vishkin apresentam um algoritmo para unificar duas listas ordenadas em tempo $O(\log(n))$ usando $O(n/\log(n))$ processadores no modelo CREW PRAM, o que é certamente ótimo. Em [204] Valiant mostra que o tempo $O(\log \log(n))$ é possível se usarmos $O(n)$ processadores, quando apenas comparações são contabilizadas. Em [34] Borodin e Hopcroft mostram que o algoritmo de Valiant pode ser implementado no model CREW PRAM, com tempo $O(\log \log(n))$ e usando $O(n)$ processadores. Em [10], Ajtai, Komlós e Szemerédi mostram que é possível ordenar n números em tempo ótimo $O(\log(n))$ usando $O(n)$ processadores no modelo EREW PRAM. Infelizmente esse resultado é apenas de interesse teórico, pois a constante envolvida na complexidade temporal é muito alta. Cole [48] forneceu um algoritmo de ordenação elegante de tempo $O(\log(n))$ para o modelo EREW PRAM usando $O(n)$ processadores, no qual as constantes são razoáveis. O algoritmo paralelo para determinar o diâmetro de um polígono convexo é uma adaptação paralela de um algoritmo de Shamos [185].

O leitor interessado em mais estudo da análise competitiva de outros algoritmos *on-line* pode consultar o livro de Borodin e El-Yaniv [33] ou o artigo de Koutsoupias e Papadimitriou [124]. O algoritmo de *cache* baseado em marcadores e o algoritmo *on-line* para leilões são discutidos no livro de Borodin e El-Yaniv, e a discussão do algoritmo de marcadores dada acima segue uma discussão similar a do livro de Motwani e Raghavan [157]. A discussão de árvores de pesquisa competitivas é baseada em um artigo de Goodrich [85]. Overmars [161] introduz o conceito de reconstrução parcial para manter estruturas de dados balanceadas. Os Exercícios C-14.11 e C-14.12 são de Sleator e Tarjan [188].

Apêndice

Fatos matemáticos úteis

A

Neste apêndice, fornecemos vários fatos matemáticos úteis. Iniciamos com algumas definições combinatórias.

Logaritmos e expoentes

A função logaritmo é definida como

$$\log_b a = c \quad \text{se} \quad a = b^c.$$

As seguintes identidades são válidas para logaritmos e expoentes:

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \ \log_b a$
4. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a/b^c = b^{a-c}$

Adicionalmente, temos

Teorema A.1: *se* $a > 0$, $b > 0$ *e* $c > a + b$, *então*

$$\log a + \log b \leq 2\log c - 2.$$

A função ***logaritmo natural*** $\ln(x) = \log_e(x)$, onde $e = 2.71828...$, é o valor da seguinte progressão:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots.$$

Adicionalmente,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

$$\ln(1+x) = x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \cdots.$$

Existem algumas desigualdades úteis relacionadas a essas funções (e que derivam dessas definições).

Teorema A.2: *se* $x > -1$,

$$\frac{x}{1+x} \le \ln(1+x) \le x.$$

Teorema A.3: *para* $0 \le x < 1$,

$$1+x \le e^x \le \frac{1}{1-x}.$$

Teorema A.4: *para quaisquer números reais positivos x e n,*

$$\left(1+\frac{x}{n}\right)^n \le e^x \le \left(1+\frac{x}{n}\right)^{n+x/2}.$$

Funções e relações inteiras

As funções "teto" e "piso" são definidas como segue:

1. $\lfloor x \rfloor$ = maior inteiro menor ou igual a x.
2. $\lceil x \rceil$ = menor inteiro maior ou igual a x.

O operador ***módulo*** é definido para inteiros $a \ge 0$ e $b > 0$ como

$$a \mod b = a - \left\lfloor \frac{a}{b} \right\rfloor b.$$

A função ***fatorial*** é definida como

$$n! = 1 \cdot 2 \cdot 3 \cdots\cdots (n-1).n$$

O coeficiente binomial é

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

que é o número de ***combinações*** diferentes que podem ser obtidas escolhendo-se k itens diferentes de uma coleção de n itens (onde a ordem dos itens não importa). O nome "coeficiente binomial" deriva da ***expansão binomial***:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k a^{n-k}.$$

Também temos as relações seguintes.

Teorema A.5: *se* $0 \le k \le n$, *então*

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \frac{n^k}{k!}.$$

Teorema A.6 (aproximação de Stirling):

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \varepsilon(n)\right),$$

onde $\varepsilon(n)$ *é* $O(1/n^2)$.

A ***progressão de Fibonacci*** é uma progressão numérica tal que $F_0 = 0$, $F_1 = 1$ e $F_n = F_{n-1} + F_{n-2}$ para $n \geq 2$.

Teorema A.7: *se F_n é definido pela progressão de Fibonacci, então F_n é $\Theta(g^n)$, onde $g = (1 + \sqrt{5})/2$ é a chamada* ***razão ideal****.*

Somatórios

Existem vários fatos úteis sobre somatórios

Teorema A.8: *fatorar somatórios:*

$$\sum_{i=1}^{n} af(i) = a\sum_{i=1}^{n} f(i),$$

desde que a não dependa de i.

Teorema A.9: *trocar de ordem:*

$$\sum_{i=1}^{n}\sum_{j=1}^{m} f(i,j) = \sum_{j=1}^{m}\sum_{i=1}^{n} f(i,j).$$

Uma forma especial de somatório é uma ***soma telescópica***:

$$\sum_{i=1}^{n}(f(i) - f(i-1)) = f(n) - f(0),$$

que freqüentemente surge na análise amortizada de uma estrutura de dados ou algoritmo.

A seguir apresentamos outros fatos sobre somatórios que surgem freqüentemente na análise de algoritmos e estruturas de dados.

Teorema A.10:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Teorema A.11:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

Teorema A.12: *se $k \geq 1$ é uma constante inteira, então*

$$\sum_{i=1}^{n} i^k \quad \text{é} \quad \Theta(n^{k+1}).$$

Outra soma comum é a ***soma geométrica***

$$\sum_{i=0}^{n} a^i,$$

para qualquer número real fixo $0 < a \neq 1$.

Teorema A.13:

$$\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a}$$

para qualquer número real fixo $0 < a \neq 1$.

Teorema A.14:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

para qualquer número real $0 < a < 1$.

Também existe uma combinação de duas formas comuns, chamada de ***soma exponencial linear***, que tem a seguinte expansão:

Teorema A.15: *para* $0 < a \neq 1$ *e* $n \geq 2$,

$$\sum_{i=1}^{n} ia^i = \frac{a-(n+1)a^{(n+1)}+na^{(n+2)}}{(1-a)^2}.$$

O n-ésimo ***número harmônico*** H_n é definido como

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$

Teorema A.16: *se* H_n *é o n-ésimo* ***número harmônico*** H_n, *então* H_n *é* $\ln(n) + \Theta(1)$.

Técnicas matemáticas úteis

Para determinar se uma função é $o()$ ou $\omega()$ de outra, às vezes é útil aplicar a seguinte regra:

Teorema A.17 (regra de L'Hôpital): *se tivermos* $\lim_{n\to\infty} g(n) = +\infty$ *e* $\lim_{n\to\infty} (n) = \infty$, *então* $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n)$, *onde* $f'(n)$ *e* $g'(n)$ *denotam as derivadas de* $f(n)$ *e de* $g(n)$.

Para derivar um limite superior e inferior para uma soma, é freqüentemente útil ***separar uma soma*** como a seguir:

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{j} f(i) + \sum_{i=j+1}^{n} f(i).$$

Outra técnica útil é ***limitar a soma por uma integral***. Se f for uma função não-decrescente, então, assumindo que os termos a seguir estão definidos,

$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx$$

Bibliografia

[1] J. D. Achter and R. Tamassia, "Selected topics in algorithms." Manuscript, 1993.

[2] G. M. Adel'son-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263-266, 1962. Tradução em inglês em *Soviet Math. Dokl.*, **3**, 1259-1262.

[3] P. K. Agarwal, "Geometric partitioning and its applications," in *Computational Geometry: Papers from the DIMACS Special Year* (J. E. Goodman, R. Pollack, and W. Steiger, eds.), American Mathematical Society, 1991.

[4] P. K. Agarwal, "Range searching," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 3 1, pp. 575-598, Boca Raton, FL: CRC Press LLC, 1997.

[5] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 3 1, pp. 1116-1127, 1988.

[6] A. V Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 255-300, Amsterdam: Elsevier, 1990.

[7] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

[9] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[10] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \log n$ parallel steps," *Combinatorica*, vol. 3, pp. 1-19, 1983

[11] S. G. Akl and K. A. Lyons, *Parallel Computational Geometry*. Prentice Hall, 1993.

[12] C. Aragon and R. Seidel, "Randomized search trees," in *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 540-545, 1989.

[13] K. Arnold and J. Gosling, *The Java Programming Language*. The Java Series, Reading, Mass.: Addison-Wesley, 1996.

[14] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions," in *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pp. 271-280, 1993.

[15] S. Arya and D. M. Mount, "Approximate range searching," in *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pp. 172-181, 1995.

[16] M. J. Atallah and D. Z. Chen, "Deterministic parallel computational geometry," in *Handbook of Computational Geometry* (J.-R. Sack and J. Urrutia, eds.), pp. 155-200, Amsterdam: Elsevier Science Publishers B.V. North-Holland, 2000.

[17] F. Aurenhammer, "Voronol diagrams: A survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, pp. 345-405, Sept. 1991.

[18] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems," in *19th ACM Symp. on Theory of Computing*, pp. 230240, 1987.

[19] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*. Reading, Mass.: Addison-Wesley, 2nd ed., 1988.

[20] E. Bach and J. Shallit, *Algorithmic Number Theory, Volume I: Efficient Algorithms*. MIT Press, 1996.

[21] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, Mass.: Addison-Wesley, 1999.

[22] O. Baruvka, "0 jistem problemu minimalnim," *Praca Moravske Prirodovedecke Spolecnosti*, vol. 3, pp. 37-58, 1926. (Em tcheco.)

[23] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," *Acta Informatica*, vol. 1, no. 4, pp. 290-306, 1972.

[24] R. Bayer and McCreight, "Organization of large ordered indexes," *Acta Inform.*, vol. 1, pp.173-189,1972.

[25] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87-90,1958.

[26] R. E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[27] J. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital signatures," in *Advances in Cryptology-EUROCRYPT 93*, vol. 765 of *Lecture Notes in Computer Science*, pp. 274-285, 1993.

[28] J. L. Bentley, "Multidimensional divide-and-conquer," *Commun. ACM*, vol. 23, no. 4, pp. 214-229, 1980.

[29] J. L. Bentley, "Programming pearls: Thanks, heaps," *Communications of the ACM*, vol. 28, pp. 245-250, 1985.

[30] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *SIGACT News*, vol. 12, no. 3, pp. 36-44, 1980.

[31] J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comput.*, vol. C-28, pp. 643-647, Sept. 1979.

[32] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1994.

[33] A. Borodin and R. EI-Yaniv, *Online Computation and Competitive Analysis*. New York: Cambridge University Press, 1998.

[34] A. Borodin and J. E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *J. Comput. Syst. Sci.*, vol. 30, no. 1, pp. 130-145, 1985.

[35] C. B. Boyer and U. C. Merzbach, *A History of Mathematics*. New York: John Wiley & Sons, Inc., 2nd ed., 1991.

[36] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, 1977.

[37] G. Brassard, "Crusade for a better notation," *SIGACT News*, vol. 17, no. 1, pp. 60-64, 1985.

[38] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[39] R. P. Brent, "Fast multiple-precision evaluation of elementary functions," *J. ACM*, vol. 23, pp. 242-251,1976.

[40] D. Bressoud and S. Wagon, *A Course in Computational Number Theory*. Key College Publishing, 2000.

[41] E. O. Brigham, *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[42] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, Mass.: Addison-Wesley, 1991.

[43] D. Burger, J. R. Goodman, and G. S. Sohi, "Memory systems," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. IS, pp. 447-461, CRC Press, 1997.

[44] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471-522, 1985.

[45] S. Carlsson, "Average case results on heapsort," *BIT*, vol. 27, pp. 2-17, 1987.

[46] V. Chvátal, "A greedy heuristic for the set-covering problem," *Math. Oper Res.*, vol. 4, pp. 233-235,1979.

[47] K. L. Clarkson, "Linear programming in $O(n3^{d^2}$ time," *Inform. Process. Lett.*, vol. 22, pp. 21-24,1986.

[48] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770-785, 1988.

[49] R. Cole, "Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm," *SIAM Journal on Computing*, vol. 23, no. 5, pp. 1075-1091, 1994.

[50] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking," *Inform. Comput.*, vol. 81, pp. 334-352, June 1989.

[51] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surv.*, vol. 11, pp. 121-137, 1979.

[52] D. Comer, *Computer Networks and Internets*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[53] S. Cook, "The complexity of theorem proving procedures," in *30th ACM Symp. on Theory of Computing*, pp. 151-158, 1971.

[54] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297-301, 1965.

[55] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[56] M. Crochemore and T. Lecroq, "Pattern matching and text compression algorithms," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 8, pp. 162-202, CRC Press, 1997.

[57] S. A. Demurjian, Sr., "Software design," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 108, pp. 2323-2351, CRC Press, 1997.

[58] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Comput. Geom. Theory Appl.*, vol. 4, pp. 235-282, 1994.

[59] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for Geometric Representations of Graphs*. Englewood Cliffs, NJ: Prentice Hall, 1998.

[60] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[61] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Commun. ACM*, vol. 31, pp. 1343-1354, 1988.

[62] H. Edelsbrunner, "A note on dynamic range searching," *Bull. EATCS*, vol. 15, pp. 34-40, 1981.

[63] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, vol. 10 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, West Germany: Springer-Verlag, 1987.

[64] J. Edmonds, "Matroids and the greedy algorithm," *Mathematical Programming*, vol. 1, pp.126-136, 1971.

[65] J. Edmonds and R. M. Karp, "Theoretical improvements in the algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, pp. 248-264, 1972.

[66] D. F. Elliott and K. R. Rao, *Fast Transform Algorithms, Analyses, and Aplications*. New York: Academic Press, 1982.

[67] Z. Emiris and V. Y. Pan, "Applications of FFT," in *Algorithms and Theory of Computation Handbook* (M. J. Atallah, ed.), ch. 17, pp. 17-1-17-30, CRC Press, 1999.

[68] S. Even, *Graph Algorithms*. Potomac, Maryland: Computer Science Press, 1979.

[69] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[70] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[71] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.

[72] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, pp. 596-615, 1987.

[73] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *SIAM Journal on Computing*, vol. 30, no. 2, pp. 209-221, 1985.

[74] R. G. Gallager, P.A. Humblet, and P.M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66-77,1983.

[75] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-3 1, no. 4, pp. 469-472, 1985.

[76] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman, 1979.

[77] A. M. Gibbons, *Algorithmic Graph Theory*. Cambridge, UK: Cambridge University Press, 1985.

[78] S. S. Godbole, "On efficient computation of matrix chain products," *IEEE Transactions on Computers*, vol. C-22, no. 9, pp. 864-866, 1973.

[79] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Reading, Mass.: Addison-Wesley, 1989.

[80] S. Golomb and L. Baumert, "Backtrack programming," *Journal of the ACM*, vol. 12, pp. 516-524, 1965.

[81] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C. Reading*, Mass.: Addison-Wesley, 1991.

[82] G. H. Gonnet and J. 1. Munro, "Heaps on heaps," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 964-971,1986.

[83] J. E. Goodman and J. O'Rourke, eds., *Handbook of Discrete and Computational Geometry*. CRC Press LLC, 1997.

[84] M. T. Goodrich, "Parallel algorithms in geometry," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 36, pp. 669-682, Boca Raton, FL: CRC Press LLC, 1997.

[85] M. T. Goodrich, "Competitive tree-structured dictionaries," in *11th ACM-SIAM Symp. on Discrete Algorithms*, pp. 494-495, 2000.

[86] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia, "Accessing the internal organization of data structures in the JDSL library," in *Proc. Workshop on Algorithm Engineering and Experimentation* (M. T. Goodrich and C. C. McGeoch, eds.), vol. 1619 of *Lecture Notes Comput. Sci.*, pp. 124-139, Springer-Verlag, 1999.

[87] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 714-723, 1993.

[88] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Inform. Process. Lett.*, vol. 1, pp. 132-133, 1972.

[89] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43-57, 1985.

[90] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*. Reading, Mass.: Addison-Wesley, 1989.

[91] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pp. 8-21, Springer-Verlag, 1978.

[92] Y. Gurevich, "What does $0(n)$ mean?," *SIGACT News*, vol. 17, no. 4, pp. 61-63, 1986.

[93] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 2nd ed., 1996.

[94] K. Hinrichs, J. Nievergelt, and P. Schorn, "Plane-sweep solves the closest pair problem elegantly," *Inform. Process. Lett.* vol. 26, pp. 255-261, 1988.

[95] D. S. Hirchsberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341-343, 1975.

[96] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, pp. 10-15, 1962.

[97] D. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems*. Boston, MA: PWS Publishers, 1996.

[98] J. E. Hopcroft and R. E. Tarjan, "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372-378, 1973.

[99] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294-303, 1979.

[100] T. C. Hu, *Combinatorial Algorithms*. Reading, Mass.: Addison-Wesley, 1981.

[101] T. C. Hu and M. T. Shing, "Computations of matrix chain products, part i," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 362-373, 1982.

[102] T. C. Hu and M. T. Shing, "Computations of matrix chain products, part ii," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 228-251, 1984.

[103] B. Huang and M. Langston, "Practical in-place merging," *Communications of the ACM*, vol. 31, no. 3, pp. 348-352, 1988.

[104] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, 1952.

[105] C. Huitema, *Routing in the Internet*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 2000.

[106] O. H. Ibarra and C. E. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems," *Journal of the ACM*, vol. 9, pp. 463-468, 1975.

[107] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, Mass.: Addison-Wesley, 1992.

[108] V. Jarnik, "0 jistem problemu minimalnim," *Praca Moravske Prirodovedecke Spolecnosti*, vol. 6, pp. 57-63, 1930. (Em tcheco.)

[109] D. S. Johnson, "Approximation algorithms for combinatorial problems," *J. Comput. Syst. Sci.*, vol. 9, pp. 256-278, 1974.

[110] R. E. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[111] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Doklady Akademii Nauk SSSR*, vol. 145, pp. 293-294, 1962. (Em russo.)

[112] D. R. Karger, P. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM*, vol. 42, pp. 321-328, 1995.

[113] R. Karp, "Reducibility among combinatorical problems of computer computaions," in *Complexity of Computer Computations* (E. Miller and J. W. Thatcher, eds.), pp. 88-104, New York: Plenum Press, 1972.

[114] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared memory machines," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), pp. 869-941, Amsterdam: Elsevier/The MIT Press, 1990.

[115] P. Kirschenhofer and H. Prodinger, "The path length of random skip lists," *Acta Informatica*, vol. 31, pp. 775-792, 1994.

[116] P. N. Klein and N. E. Young, "Approximation algorithms," in *Algorithms and Theory of Computation Handbook* (M. J. Atallah, ed.), ch. 34, pp. 34-1-34-19, CRC Press, 1999.

[117] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1st ed., 1968.

[118] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 2nd ed., 1973.

[119] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973.

[120] D. E. Knuth, "Big omicron and big omega and big theta," in *SIGACT News*, vol. 8, pp. 18-24, 1976.

[121] D. E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 3rd ed., 1998.

[122] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 323-350, 1977.

[123] N. Koblitz, *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.

[124] E. Koutsoupias and C. H. Papadimitriou, "On the *k*-server conjecture," *Journal of the ACM*, vol. 42, no. 5, pp. 971-983, 1995.

[125] E. Kranakis, *Primality and Cryptography*. John Wiley and Sons, 1986.

[126] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," *IEEE Trans. Comput.*, vol. C-34, pp. 965-968, 1985.

[127] J. B. Kruskal, Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer Math. Soc.*, vol. 7, pp. 48-50, 1956.

[128] D. T. Lee, "Computational geometry," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 6, pp. 111-140, CRC Press, 1997.

[129] D. T. Lee and F. P. Preparata, "Computational geometry: a survey," *IEEE Trans. Comput.*, vol. C-33, pp. 1072-1101, 1984.

[130] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, CA: Morgan-Kaufmann, 1992.

[131] L. A. Levin, "Universal sorting problems," *Problemy Peredachi Informatsii*, vol. 9, no. 3, pp. 265-266, 1973. (Em russo.)

[132] A. Levitin, "Do we teach the right algorithm design techniques?," in *30th ACM SIGCSE Symp. on Computer Science Education*, pp. 179-183, 1999.

[133] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*. Upper Saddle River, New Jersey: Prentice-Hall, 2nd ed., 1998.

[134] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, Mass.: AddisonWesley, 1997.

[135] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, Mass./New York: The MIT Press/McGraw-Hill, 1986.

[136] L. Lovász, "On the ratio of optimal integral and fractional covers," *Discrete Math.*, vol. 13, pp. 383-390, 1975.

[137] N. A. Lynch, *Distributed Algorithms*. San Francisco: Morgan Kaufmann, 1996.

[138] J. Matoušek, "Geometric range searching," *ACM Comput. Surv.*, vol. 26, pp. 421-461, 1994.

[139] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of Algorithms*, vol. 23, no. 2, pp. 262-272, 1976.

[140] E. M. McCreight, "Priority search trees," *SIAM J. Comput.*, vol. 14, no. 2, pp. 257-276, 1985.

[141] C. J. H. McDiarmid and B. A. Reed, "Building heaps fast," *Journal of Algorithms*, vol. 10, no.3, pp. 352-365, 1989.

[142] C. C. McGeoch, "Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups," *ACM Computing Surveys*, vol. 24, no. 2, pp. 195-212, 1992.

[143] C. C. McGeoch, "Toward an experimental method for algorithm simulation," *INFORMS Journal on Computing*, vol. 8, no. 1, pp. 1-15, 1996.

[144] C. C. McGeoch, D. Precup, and P. R. Cohen, "How to find the Big-Oh of your data set (and how not to)," in *Advances in Intelligent Data Analysis*, vol. 1280 of *Lecture Notes in Computer Science*, pp. 41-52, Springer-Verlag, 1997.

[145] N. Megiddo, "Linear-time algorithms for linear programming in R^3 and related problems," *SIAM J. Comput.*, vol. 12, pp. 759-776, 1983.

[146] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *J. ACM*, vol. 3 1, pp. 114-127, 1984.

[147] K. Mehlhorn, "A best possible bound for the weighted path length of binary search trees," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 235-239, 1977.

[148] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.

[149] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, vol. 2 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.

[150] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, vol. 3 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.

[151] K. Mehlhorn and S. Näher, *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge, UK: Cambridge University Press, 1999.

[152] K. Mehlhorn and A. Tsakalidis, "Data structures," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 301-341, Amsterdam: Elsevier, 1990.

[153] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. Symp. on Security and Privacy*, IEEE Computer Society Press, 1980.

[154] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology–CRYPTO '89* (G. Brassard, ed.), vol. 435 of *Lecture Notes in Computer Science*, pp. 218-238, Springer-Verlag, 1990.

[155] M. H. Morgan, *Vitruvius: The Ten Books on Architecture*. New York: Dover Publications, Inc., 1960.

[156] D. R. Morrison, "PATRICIA–practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514-534, 1968.

[157] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY: Cambridge University Press, 1995.

[158] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Reading, Mass.: Addison-Wesley, 1996.

[159] R. Neapolitan and K. Naimipour, *Foundations of Algorithms Using C++ Pseudocode*. Boston: Jones and Bartlett Publishers, 1998.

[160] J. O'Rourke, *Computational Geometry in C*. Cambridge University Press, 1994.

[161] M. H. Overmars, *The Design of Dynamic Data Structures*, vol. 156 of *Lecture Notes Comput. Sci.* Heidelberg, West Germany: Springer-Verlag, 1983.

[162] J. Pach, ed., *New Trends in Discrete and Computational Geometry*, vol. 10 of *Algorithms and Combinatorics*. Springer-Verlag, 1993.

[163] T. Papadakis, J. I. Munro, and P. V. Poblete, "Average search and update costs in skip lists," *BIT*, vol. 32, pp. 316-332, 1992.

[164] C. H. Papadimitriou and K. Steiglitz, "Some complexity results for the traveling salesman problem," in *Proc. 8th Annu. ACM Sympos. Theory Comput.*, pp. 1-9, 1976.

[165] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice Hall, 1982.

[166] D. Peleg, *Distributed Computing: A Locally-Sensitive Approach*. Philadelphia: SIAM, 2000.

[167] P. V. Poblete, J. I. Munro, and T. Papadakis, "The binomial transform and its application to the analysis of skip lists," in *Proceedings of the European Symposium on Algorithms* (*ESA*), pp. 554-569, 1995.

[168] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York, NY: Springer-Verlag, 1985.

[169] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, vol. 36, pp. 1389-1401, 1957.

[170] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668-676, 1990.

[171] M. O. Rabin, "A probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, 1980.

[172] J. H. Reif, *Synthesis of Parallel Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.

[173] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120-126, 1978.

[174] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. on Computing*, vol. 6, pp. 563-581, 1977.

[175] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Reading, MA: Addison-Wesley, 1990.

[176] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[177] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[178] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15, no. 1, pp. 76-100, 1993.

[179] B. Schneier, *Applied cryptography: protocols, algorithms, and sourcecode in C*. New York: John Wiley and Sons, Inc., 1994.

[180] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley and Sons, Inc., second ed., 1996.

[181] A. Schönhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," *Computing*, vol. 7, no. l, pp. 37-44, 1971.

[182] R. Sedgewick, *Algorithms*. Reading, MA: Addison-Wesley, 1st ed., 1983.

[183] R. Sedgewick, *Algorithms in C++*. Reading, MA: Addison-Wesley, 1992.

[184] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*. Reading, Mass.: Addison-Wesley, 1996.

[185] M. I. Shamos, "Geometric complexity," in *Proc. 7th Annu. ACM Sympos. Theory Comput.*, pp. 224-233, 1975.

[186] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms*, vol. 2, pp. 88-102, 1981.

[187] M. Sipser, *Introduction to the Theory of Computation*. PWS Publishing Co., 1997.

[188] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, pp. 202-208, 1985.

[189] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652-686, 1985.

[190] R. Solovay and V. Strassen, "A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 84-85, 1977.

[191] R. Solovay and V. Strassen, "Erratum: A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, vol. 7, no. 1, 1978.

[192] W. Stallings, *Cryptography and network security: principles and practice*. Upper Sadle River, NJ 07458, USA: Prentice-Hall, Inc., second ed., 1999.

[193] G. A. Stephen, *String Searching Algorithms*. World Scientific Press, 1994.

[194] R. Tamassia, "Graph drawing," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 44, pp. 815-832, Boca Raton, FL: CRC Press LLC, 1997.

[195] R. Tamassia, M. T. Goodrich, L. Vismara, M. Handy, G. Shubina, R. Cohen, B. Hudson, R. S. Baker, N. Gelfand, and U. Brandes, "JDSL: The data structures library in Java," *Dr. Dobb's Journal*, vol. 323, April 2001.

[196] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[197] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, vol. 14, pp. 862-874, 1985.

[198] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146-160, 1972.

[199] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.*, vol. 18, pp. 110-127, 1979.

[200] R. E. Tarjan, *Data Structures and Network Algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.

[201] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306-318, 1985.

[202] G. Tel, *Introduction to Distributed Algorithms*. New York: Cambridge University Press, 1994.

[203] J. D. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1983.

[204] L. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, no. 3, pp. 348-355, 1975.

[205] J. van Leeuwen, "Graph algorithms," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 525-632, Amsterdam: Elsevier, 1990.

[206] J. S. Vitter, "Efficient memory access in large-scale computation," in *Proc. 8th Sympos. Theoret. Aspects Comput. Sci.*, Lecture Notes Comput. Sci., Springer-Verlag, 1991.

[207] J. S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in *Algorithms and Complexity* (J. van Leeuwen, ed.), vol. A of *Handbook of Theoretical Computer Science*, pp. 431-524, Amsterdam: Elsevier, 1990.

[208] J. Vuillemin, "A unifying look at data structures," *Commun. ACM*, vol. 23, pp. 229-239, 1980.

[209] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11-12, 1962.

[210] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347-348, 1964.

[211] D. Wood, *Data Structures, Algorithms, and Performance*. Reading, Mass.: Addison-Wesley, 1993.

[212] F. F. Yao, "Computational geometry," in *Algorithms in Complexity* (R. A. Earnshaw and B. Wyvill, eds.), pp. 345-490, Amsterdam: Elsevier, 1990.

[213] C. K. Yap, *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 1999.

Índice

δ-aproximação, 608-609
c-incremental, 584
2SAT, 598, 629-630
3SAT, 598, 599

A

aceitação de *string*, 585
Achter, 502-503
acíclico, 318-319
Adel'son-Vel'skii, 219-220
adivinhe-e-teste, 270-272
adjacente, 294
Adleman, 470-471
Aggarwal, 672
agrupamento secundário, 131-132
Aho, 145-146, 219-220, 261, 445-446, 502-503, 632
Ahuja, 340-341, 379, 411-412
Ajtai, 672
Akl, 672
alfabeto, 417-418
algoritmo, 20
algoritmo da peneira, 144-145
algoritmo de Barůvka, 369-372, 520-522
algoritmo de Bellman-Ford, 351-354
algoritmo de Dijkstra, 345-351, 372-376
algoritmo de Edmonds-Karp, 392-394
algoritmo de Euclides, 450-453
  binário, 452-453
  estendido, 459-460
algoritmo de Floyd-Warshall, 322-323, 340-341
algoritmo de Ford-Fulkerson, 386-394
algoritmo de Kruskal, 363-367
algoritmo de marcação e varredura, 324-326
algoritmo de memória externa, 635-646
algoritmo de Prim-Jarník, 366-368
algoritmo de Rabin-Miller, 464-465
algoritmo de Solovay-Strassen, 461-462
algoritmo de Strassen, 275-276
algoritmo de varredura de Graham, 570-573
algoritmo do estado de ligação, 525-527
algoritmo do vetor de distâncias, 524-525
algoritmo genérico de junção, 230
algoritmo *on-line*, 655-667
algoritmo para leilão, 662-664
algoritmo paralelo, 646-655
algoritmos de aproximação, 608-616
algoritmos de *cache*, 656-662
algoritmos de segurança, 474-481
algoritmos distribuídos, 507-510
  modelo de troca de mensagens, 507-509
altura, 90-92
amortização, 48-55, 91-92, 140-141, 194-198, 218-219, 231-239, 424-425, 614-616, 664-667
  função potencial, 50-52
  método de contabilização, 49-51
análise de algoritmos, 23-48
  caso médio, 26-27
  pior caso, 26-27
ancestral, 86-87, 317-318
anti-horário, 564-565
antípoda, 654
aproximação de Stirling, 673-674
Aragon, 580-581
arco, 293
aresta, 293, 563-564
  destino, 294
  grau de entrada, 294
  grau de saída, 294
  incidente, 294
  múltipla, 295
  origem, 294
  paralela, 295
  vértice final, 294
aresta de árvore, 320-321, 322-323
aresta de cruzamento, 317-318, 320-323
aresta de descoberta, 307, 309, 317-318, 320-323
aresta de retorno, 307, 309, 320-323, 338-339
aresta de separação, 310-311
aresta fora da árvore, 320-323
Ariadne, 292
aritmética modular, 74-75, 133-134, 449-450, 453-457, 673-674
Arnold, 145-146
Arquimedes, 20, 66-67
arquivo de *log*, 123-124, 129
arranjo de *buckets*, 124-125
árvore (2,4), 169-175
  propriedade da profundidade, 169-170
  propriedade do tamanho, 169-170
árvore, 86-146, 296
  altura, 90-92
  binária, *veja* árvore binária
  estrutura encadeada, 103-104
  genérica, 165-169
  multidimensional, 540-555
  nível, 95
  nodo externo, 86-87
  nodo filho, 86-87
  nodo interno, 86-87
  nodo pai, 86-87
  nodo raiz, 86-87
  nodo, 86-87

ordenada, 87-88
profundidade, 90-92
tipo abstrato de dados, 88-90
árvore (*a*, *b*), 640-642
propriedade de profundidade, 640
propriedade de tamanho, 640
árvore AVL, 159-165, 209, 559-560
fator de balanceamento, 209
propriedade da altura balanceada, 159
árvore-B, 642-643
árvore BFS, 317-318
árvore binária, 87-88, 95-103, 224
completa, 108-109
estrutura encadeada, 102-103
filho direito, 87-88
filho esquerdo, 87-88
nível, 95
própria, 87-88
representação por vetor, 100-102
árvore de busca por prioridade, 547-551
árvore de cobertura, 296-297, 305-307, 309-310, 316-319, 361-362
árvore de cobertura mínima, 361-372, 520-521
algoritmo de Barůvka, 369-372, 520-522
algoritmo de Kruskal, 363-367, 371-372
algoritmo de Prim-Jarník, 366-368, 371-372
árvore de decisão, 243-244
árvore de partições, 552-553
árvore de pesquisa balanceada, 168-169
árvore de pesquisa binária, 152-158
inserção, 155-156
reestruturação trinodo, 162
remoção, 156-158
rotação, 162
árvore de pesquisa competitiva, 664-667
árvore de pesquisa genérica, 165-166
árvore de profundidade limitada, 165-166
árvore de recursão, 269-270
árvore DFS, 307, 309
árvore DFS dirigida, 320-321
árvore intervalar, 540-553
árvore *k*-D, 554-555
árvore vermelho-preta, 175-188, 212, 559-560
propriedade da raiz, 175-176
propriedade de profundidade, 175-176
propriedade externa, 175-176
propriedade interna, 175-176
troca de cores, 179, 182
árvores com balanceamento de energia, 664-667
árvores de Steiner, 529-532
árvore intervalar por prioridade, 551-553
Arya, 580-581
assinatura digital, 465-466
El-Gamal, 473-474
RSA, 471-472
associação, 395
atingibilidade, 318-319
atribuição máxima bipartida, 395-396
Attalah, 672
aumentos de Edmonds-Karp, 392
Aurenhammer, 580-581
autoridade certificadora, 479-480
Awerbuch, 439-440

B

Baase, 502-503
backtracking, 305-307, 616-621, 632
Baeza-Yates, 219-220, 261, 445-446, 672
Barůvka, 379
Baumert , 632
Bayer, 219-220, 672
Bellman, 287-288, 379
Bentley, 145-146, 287-288, 580-581
best-first search, 621-622
BFS, *veja* caminhamento em largura
biconexa, 310-311
bit commitment, 476-477
blocking, 636-637
bloco de uso único, 468
Booch, 145-146
bootstraping, 166-168
Borodin, 672
Boyer, 66-67, 445-446
branch-and-bound, 621-632
Brassard, 66-67, 632
Bratley, 632
Brent, 672
Brigham, 502-503
bubble-sort, 145-146
bubbling-up, 111-112
bucket-sort, 245-247
Budd, 145-146
Burger, 672

C

cabeçalho, 82
cache, 635
caminhamento em árvore, 92-95, 97-100
genérico, 98-100
interfixado, 98, 153-154, 156-158, 162
passeio de Euler, 98-100
por nível, 141-142
pós-fixado, 93-95, 97
prefixado, 92-93, 97
caminhamento em largura, 316-319, 322-323
caminhamento em profundidade, 305-310
caminho, 296
comprimento, 344
dirigido, 296
peso, 344
simples, 296
caminho aumentado, 386-387
caminho mínimo, 344-361
algoritmo de Bellman-Ford, 351-354
algoritmo de Dijkstra, 345-351, 372-376
multiplicação de matrizes, 356-361
capacidade de aresta, 382
capacidade residual, 386
Cardelli, 145-146

carimbos de tempo, 475-476
Carlsson, 145-146
certificado, 479-480, 587
chamada por valor, 73
chave, 104-105, 122-124, 165-166
Chen, 672
Chvátal, 632
ciberdólar, 49-50
ciclo, 296
  dirigido, 296, 318-319
ciclo aumentado, 397
CICLO-HAMILTONIANO, 588, 606, 607-608, 630-632
CIRCUIT- SAT, 588-592, 595-596
circuito booleano, 588
circuito de Euler, 337-338, 340-341
circuito de Euler percorrido, 98, 145-146
Clarkson, 261
classe complexa, 586
classe de equivalência, 311-312
classificação em lista, 650-651
cláusulas, 595-596
clique, 601
CLIQUE, 601, 627-628
CNF, *veja* forma normal conjuntiva
CNF-SAT, 595-598, 617-619, 629-630, 632
cobertura convexa, 562-573, 580-581
  algoritmo de varredura de Graham, 570-573
  embrulho para presente, 568-570
cobertura de vértices, 588-589
codificação unária, 584
código de Huffman, 435-437
código de prefixos, 435-436
código *hash*, 125-126, 126-127
Cole, 445-446, 672
coleta de lixo, 324-327
colinear, 564-565
colocação (*rank*), 149-150, 234-235
Comer, 535-536, 672
comparador, 105, 222, 568-570, 573
  radial, 568-571, 573
complemento, 586
componente biconexo, 310-311
componentes conexos, 296, 309-310, 317-318
componentes de ligação, 312-314
composto, 449
compressão de caminho, 233-234
compressão de texto, 435-437
comprimento de caminho, 141-142
computação distribuída
  modelo assíncrono, 507-508
  modelo síncrono, 507-508
confidencialidade, 448
congruência, 449-450
congruente, 449-450
conjunto independente, 287-288
conjunto máximo independente, 340-341
conjunto, 229-239
co-*NP*, 586, 588
contador de programa, 73
contêiner, 145-146
contêiner de incidência, 302-303
contradição, 39-40
contrapositivo, 39-40
convexidade, 563-564, 566-567
convolução, 481-482
Cook, 632
Cooley, 502-503
coordenadas, 540-541
coordenadas cartesianas, 562-563
Cormen, 219-220, 340-341, 379, 411-412, 502-503, 632
corte, 384
corte mínimo, 385
criptografia, 465-474
  sistema de criptografia El-Gamal, 473-474
  sistema de criptografia RSA, 470-472
criptografia de chave pública, 466-467, 469-474
criptografia simétrica, 467-468
criptossistema de chave pública, 469
criptossistema El-Gama, 473-474
criptossistema RSA, 470-472
Crochemore, 445-446
cursor, 82
curva à direita, 564-565
custo, 397

D

decomposição em primos, 449
decriptação, 465-466
Demurjian, 145-146
descendente, 86-87, 317-318
desigualdade triangular, 613
destino, 294, 381, 382
DFS, *veja* caminhamento em profundidade
DFT, *veja* transformada discreta de Fourier
diagrama de Voronoi, 579-580
diâmetro, 142-144, 515-516, 654
DiBattista, 145-146, 340-341
dicionário, 122-135, 148-220
  arquivo de *log*, 123-124
  árvore (2, 4), 169-175
  árvore AVL, 159-165
  árvore de pesquisa binária, 152-158
  árvore vermelho-preta, 175-188
  não-ordenado, 122-124
  operações de atualização, 155-157, 160-164, 201-203
  ordenado, 122-123, 148-149
  skip list, 198-206
  tabela de pesquisa, 149-153
  tabela *hash*, 124-135
  tipo abstrato de dados, 122-124, 148-149
dicionário autenticado, 475-476
dicionário bidimensional, 540-541
dígrafo, 318-319
Dijkstra, 379
dilema do locatário, 655-656
distância, 344
distância de edição, 444-445
distância residual, 392
divisão e conquista, 223-225, 239-240, 267-277, 648-649, *veja também* poda e busca

DNF, *veja* forma normal disjuntiva
DNF-DISSAT, 627-628
DNF-SAT, 627-628
d-nodo, 165-166
down-heap bubbling, 112-114, 117-118
duplicação recursiva, 650-651

E

Edelsbrunner, 580-581
Edmonds, 287-288, 411-412
eleição de líder, 510-516
Elliott, 502-503
El-Yaniv, 672
embaralhamento inverso, 494-495
embrulho para presente, 568-570
Emiris, 502-503
encadeamento separado, 129
endereçamento aberto, 131-132
envelope superior, 286-287
escalonamento de máquina, 286-287
escalonamento de tarefas, 265-266
escolha gulosa, 264, 437-438
espaço amostral, 42-43
espaço de probabilidade, 42-43
esquema de aproximação de tempo polinomial, 609-610
estatística de ordem, 249-250
estável, 246-247
estratégia de marcadores, 659-660
estrutura de dados, 20
  secundária, 166-168
estrutura encadeada, 102-103
Even, 340-341, 379, 411-412
evento, 42-43
expansão binomial, 673-674
expoente, 37-38
exponenciação modular, 457-459
exponencial linear, 673

F

fator de balanceamento, 209
fator de carga, 130
fatorial, 673-674
fechamento, 358-359
fechamento de matriz, 358-361
fechamento transitivo, 318-319, 321-322
FFT, *veja* transformada rápida de Fourier
FIFO, 73-74, 645-646
fila, 73-77
  implementação por arranjo, 73-76
  tipo abstrato de dados, 73-74
fila de prioridades, 104-122, 222
  implementação por *heap*, 109-114, 116
  implementação por seqüência, 106-108
filho à direita, 87-88
filho esquerdo, 87-88
filhos, 86-87
first-in, *first-out*, 73-74
Flajolet, 66-67
floresta, 296
floresta de cobertura, 309-310, 317-318
Floyd, 145-146
fluxo em redes, 382
folhas, 86-87
fonte, 381, 382
força bruta, 417-418
Ford, 379
forma de coeficientes, 481-482
forma normal conjuntiva, 595-596, 598
forma normal de Chomsky, 287-288
forma normal disjuntiva, 597
fórmula fechada, 27-28
fortemente conexo, 318-319
fortemente ***NP***-difícil, 602-603
fortemente resistente a colisões, 474-475
fraca, 507-508
frame, 72
Fulkerson, 411-412
função de Ackermann, 238-239, 261
função de falha, 422-423
função *hash*, 125-126, 131-134
  2-universal, 132-133
  de uma via, 474-475
função objetivo, 264
função potencial, 50-52
função testemunha de composição, 460-461
função totiente, 455-456
fusão, 172-174, 640-643

G

Gabow, 261
GAD, *veja* grafo dirigido acíclico
Galleger, 535-536
Garey, 632
gargalo, 394
Gauss, 36-37
Gavril, 632
GCD, *veja* maior divisor comum
geometria computacional, 540-581
  cobertura convexa, 562-573
  degenerações, 565-566, 573
  intersecção de segmentos, 556-559
  orientação, 564-566
  pares mais próximos, 558-561
  plano rotacional, 556-561
  proximidade, 558-559
  representações, 562-564
gerador, 457
gerador de números pseudo-aleatórios, 198-199
gerenciamento de memória, 324-326, 635-638, 656-662
Gibbons, 340-341, 379, 411-412
Godbole, 287-288
Goldberg, 145-146
Golomb, 632
Gonnet, 145-146, 219-220, 261, 672
Goodman, 580-581
Goodrich, 580-581, 672
googol, 32
Gosling, 145-146

grafo, 292-341, 343-379
  acíclico, 318-319
  atingibilidade, 318-320, 322-324
  bipartido, 395
  caminhamento em largura, 316-319, 320-323
  caminhamento em profundidade, 305-310, 320-323
  caminhos mínimos, 322-324
  conexo, 296, 317-318
  denso, 323-324
  dígrafo, 318-319
  dirigido, 293, 294, 318-330
    acíclico, 327-330
    fortemente conexo, 318-319
  estruturas de dados, 299-305
    lista de adjacência, 302-305
    matriz de adjacência, 303-305
    lista de arestas, 299-302
  métodos, 296-299
  misto, 294
  não-dirigido, 293, 294
  ponderado, 343-379
  simples, 295
  tipo abstrato de dados, 293-299
grafo residual, 390-391
Graham, 66-67, 379, 580-581
gramática, 287-288
grau, 294, 481-482
grau de entrada, 294
grau de saída, 294
grupo, 456-457
grupo multiplicativo, 456-457
grupos de colocação (*rank*), 235-237
guardas da galeira de arte, 286-287
Guibas, 219-220
Guttag, 145-146

H

Haken, 287-288
hashing duplo, 131-132
hashing universal, 132-135
heap unificável, 218-219
heap, 108-120
  construção *bottom-up*, 117-120
heapsort, 116-120, 232
Hell, 379
Hennessy, 672
heurística do espelho, 420
heurística do salto de caracteres, 420
hierarquia de memória, 635
hierárquico, 69
Hinrichs, 580-581
Hirchsberg, 287-288
Hoare, 261
Hochbaum, 632
Hopcroft, 145-146, 219-220, 261, 340-341, 502-503, 632, 672
Hu, 287-288
Huang, 261
Huffman, 445-446
Huitema, 535-536

I

Ibarra, 632
incidente, 294
independente, 42-45
INDEPENDENTE-SET, 629-630
índice, *veja* logaritmo discreto
indução, 39-41
in-place, 252, 326-327
insertion-sort, 107-108, 222
integridade de dados, 448, 466-467
inteiros grandes, 273-274
intersecção de segmentos, 556-559
invariantes de laços, 41-42
inversão, 258-259
inverso aditivo, 453-454
inverso multiplicativo, 453-454, 459
inverso multiplicativo modular, 459
irmão, 86-87
itens, 122-123
iterador, 85-86

J

JáJá, 145-146, 672
Jarník, 379
JDSL, 145-146
jogar uma moeda, 476-477
Johnson, 632
Jones, 340-341

K

Karatsuba, 287-288
Karger, 379
Karp, 145-146, 411-412, 632
Kim, 632
Klein, 379, 632
KNAPSACK (mochila), 605, 609-612, 623, 628-632
Knuth, 66-67, 145-146, 219-220, 261, 340-341, 445-446, 672
Komlós, 672
Kosaraju, 340-341
Koutsoupias, 672
Kruskal, 379, 672

L

Landis, 219-220
Langston, 261
last-in, *first-out*, 70-71
LCS, *veja* maior subseqüência comum
Lecroq, 445-446
LEDA, 145-146
Lee, 580-581
Leighton, 672
leis de De Morgan, 39-40
Leiserson, 219-220, 340-341, 379, 411-412, 502-503, 632
Levin, 632
Lewis, 632
lexicográfica, 246-247
LIFO, 70-71
limitação randomizada, 663-664

limites de Chernoff, 257-258
Lindholm, 145-146
linearidade do valor esperado, 43-44, 250-251
linguagem, 585
linha, 562-563
linha de *cache*, 636-637
Liskov, 145-146
lista, 80-84, 123-124
  tipo abstrato de dados, 81-84
lista de adjacência, 299-300, 302-303
lista de arestas, 299-302
lista de certificados revogados, 479-480
lista de controle de acesso, 466-467
lista encadeada,
  duplamente encadeada, 82-85
literais, 595-596
localidade de referência, 636-637
logaritmo, 37-38, 673
  natural, 673
logaritmo discreto, 473
Lovász, 632
LRU, 645-646
Lynch, 535-536
Lyons, 672

M

Magnanti, 340-341, 379, 411-412
maior divisor comum, 449-453
  algoritmo de Euclides, 450-453
maior subseqüência comum, 438-441
mapa de compressão, 125-128
máquina de acesso aleatório, 24-25, 634
Máquina Virtual Java, 76-77, 145-146
marcação e varredura, 324-326
matriz de adjacência, 299-300, 303-305
matriz de incidência, 602-603
matriz identidade, 485-486
McCreight, 445-446, 580-581, 672
McDiarmid, 145-146
McGeoch, 66-67
mecanismo de busca, 229, 434-435
média de prefixos, 45-46
mediana, 249-250
Megiddo, 261
Mehlhorn, 219-220, 340-341, 379, 411-412, 580-581, 672
memória de núcleo, 635
memória externa, 634-646, 672
memória interna, 635
memória principal, 635
memória virtual , 636-637
merge-sort, 223-229
  árvore, 224
  múltiplas entradas, 643-646
Merzbach, 66-67
método de contabilização, 49-51
método de divisão, 127-128
método de Horner, 64-65
método guloso, 264-267, 344, 345, 437-438
método mestre, 272-274
Minotauro, 292
modelo de passagem de mensagens, 507-509
modelo PRAM, 646-647
  CRCW, 647-648
  CREW, 647-648
  EREW, 647-648
módulo, 449-450
Moore, 445-446
Morris, 445-446
Morrison, 445-446
Motwani, 219-220, 261, 672
Mount, 580-581
MST, *veja* árvore de cobertura mínima
multiplicação inteira, 273-276
multiprogramação, 75-76
Munro, 145-146
mutuamente independentes, 42-43

N

Naimipour, 632
n-cast, 535-536
Neapolitan, 632
  algoritmo de Edmonds-Karp, 392-394
  algoritmo de Ford-Fulkerson, 386-394
  aresta de avanço, 384
  aresta de retorno, 394
  caminho aumentado, 384
  capacidade da aresta, 382
  capacidade do corte, 385
  capacidade residual, 386
  ciclo aumentado, 386-387
  corte mínimo, 385
  corte, 384-385
  distância residual, 392
  fluxo através de um corte, 385
  fluxo de custo mínimo, 397-403-404
  fluxo em redes, 397
  gargalo, 382
  grafo residual, 390-391
  problema do fluxo máximo, 383, 386
  rede de fluxo, 382-384
  regra da capacidade, 382
  regra da conservação, 382
  teorema de fluxo máximo e corte mínimo, 387-388
  valor de fluxo, 383
nível, 95, 316
nodo, 80-81, 86-89, 293
  ancestral, 86-87
  balanceado, 160-161
  borda, 542-543
  descendente, 86-87
  externo, 542-543
  externo, 86-87
  filho, 86-87
  interno, 542-543
  interno, 86-87
  irmão, 86-87
  não-balanceado, 160-161
  pai, 86-87

raiz, 86-87
redundante, 429-430
tamanho, 194-195
nodo-limite, 542-543
notação assintótica, 28-48
O(), 28-31, 66-67
o(), 33
Θ (), 31
Ω (), 31
ω (), 33
NP, 586, 587
NP-completude, 583-632
NP-difícil, 590-591
numeração de nível, 100-101
número de Catalan, 278-279
número harmônico, 673
números de Carmichael, 460-461

O

O'Rourke, 580-581
objetos ativos, 324-326
objetos raiz, 324-326
Ofman, 287-288
operações primitivas, 24-28, 49-50
ordem total, 104-105
ordenação, 106, 222-229, 239-248
bucket-sort, 245-247
estável, 246-247
fila de prioridades, 106
heapsort, 116-120
in-place, 117-118, 252
insertion-sort, 107-108
limite inferior, 243-245
memória externa, 643-646
merge-sort, 223-229
quick-sort, 239-243
radix-sort, 246-248
selection-sort, 106-107
ordenação topológica, 327-330
orientação, 564-566
origem, 294
Orlin, 340-341, 379, 411-412
otimalidade de subproblemas, 278-279, 281-282
Ottmann, 580-581
overflow, 170-171, 642-643
Overmars, 672

P

Pach, 580-581
padrão de projeto
amortização, 52-55
comparador, 105
decorador, 331-334
divisão e conquista, 223-225, 239-240, 267-277
força bruta, 417-418
padrão do método do modelo, 334
padrão localizador, 120-122
pai, 86-87
palíndromo, 444-445
Pan, 580-581
Papadimitriou, 287-288, 632, 672
pares mais próximos, 558-561
partição, 231-239
partição de arranjos, 652
PARTITION, 629-630
Patashnik, 66-67
Patterson, 672
Peleg, 535-536
pequeno teorema de Fermat, 454-455
pesquisa à frente, 200-201
pesquisa binária, 149-153
pesquisa de padrões, 417-426
algoritmo de Boyer-Moore, 420-423
algoritmo de Knuth-Morris-Pratt, 422-426
força bruta, 417-420
pesquisa intervalar
de três lados, 547-548
em uma dimensão, 541-544
pilha, 70-73
implementação usando arranjos, 70-72
tipo abstrato de dados, 70-71
pilha de métodos, 72-73
pilha de protocolos em rede, 506-507
camada de aplicação, 506
camada de *data-link*, 506
camada de rede, 506
camada de transporte, 506
camada física, 506
poda e busca, 249-252
polígono, 562-564, 566-567
arestas, 563-564
convexo, 563-564
polinomial, 64-65
sem intersecção, 563-564
simples, 563-564
vértices, 563-564
ponto, 562-563
ponto de ancoragem, 568-572
pontos finais, 294
posição, 80-81, 88-89, 199-200
postulado de Bertrand, 133-134
Pratt, 445-446
prefixo, 417
prefixo paralelo, 651-652
Preparata, 581
preto duplo, 182
Prim, 379
primo, 449
primo relativo, 449-450, 453-454
probabilidade, 42-45
probabilidade condicional, 43-44
problema da mochila, 264-265, 281-285
problema de decisão, 585
problema de Josephus, 140-141
problema de otimização, 281-282, 585
problema do caixeiro viajante, 345, 607-608
produto de matrizes, 277-281
profundidade, 90-92
programação dinâmica, 277-285, 322-323, 439-441
iterador, 85-86

localizador, 120-122
método guloso, 437-438
métodos modelo, 334
poda e busca, 249-252
posição, 80-81
progressão de Fibonacci, 675
projeto de componentes, 593-594
projeto orientado a objetos, 145-146
propriedade anti-simétrica, 104-105
propriedade da altura/balanceamento, 159-164
propriedade de ordem do *heap*, 108-109
propriedade reflexiva, 104-105, 311-312
propriedade simétrica, 311-312
propriedade transitiva, 104-105, 311-312
proximidade, 558-559
pseudoblocos, 645-646
pseudocódigo, 22-24
pseudoprimo de Euler, 463-464
PTAS, *veja* esquema de aproximação de tempo polinomial
Pugh, 219-220

Q

quadrados sucessivos, 458
quadtree, 552-554
quick-select randomizado, 249-250
quick-sort, 238-243
árvore, 239-240
quick-sort randomizado, 240-242

R

radix-sort, 246-248
Raghavan, 219-220, 261, 672
raiz, 86-87
raiz primitiva, 457
raiz primitiva da unidade, 482-483
RAM, *veja* máquina de acesso aleatório
Ramachandram, 145-146
randomização, 198-200
Rao, 502-503
razão ideal, 675
recursão, 27-28, 73
redutível em tempo polinomial, 590-591
Reed, 145-146
reescalável, 609-610
reestruturação trinodo, 162, 178-179
regra da capacidade, 382
regra da conservação, 382
regra de Horner, 481-482
regra de L'Hôpital, 673
rehash, 130
Reif, 672
relação assimétrica, 293
relação de equivalência, 311-312
relação de ligação, 311-312
relação de recorrência, 27-28, 228-229, 267-274
relação simétrica, 293
relaxamento, 345-346, 351-352
resíduo, 453-454
resíduo quadrático, 461-462
resistência a colisões, 474-475
resolução de colisões, 124-125, 128-135
restrição, 593-594
Ribeiro-Neto, 445-446
Rivest, 219-220, 340-341, 379, 411-412, 470-471, 502-503, 632
Robson, 145-146
Rosenkrantz, 632
rotação, 162, 162-164
dupla, 162
simples, 162
roteador, 522
roteadores IP, 507
roteamento
algoritmo do vetor de distâncias, 524-525
algoritmo *link-state*, 525-527
árvores centradas, 528-530
broadcast, 522-523
caminhamento reverso, 527-528
multicast, 526-532
unicast, 524-527
round-robin, 76-77
RPF, *veja* roteamento, caminhamento reverso
RSA, 470-471

S

Samet, 580-581, 672
SAT, 627-628
satisfação de atribuição, 588-589
Saxe, 287-288
Schaffer, 145-146
Schönhage, 287-288
Sedgewick, 66-67, 145-146, 219-220, 502-503, 580-581
segmento, 562-564
segmentos ortogonais, 556
Seidel, 580-581
seleção, 249-252
selection-sort, 106-107, 222
self-loop, 295
semente, 198-199
senha, 466-467
sensível à saída, 556, 568-570
sentido horário, 564-565
sentinela, 82, 123-124
seqüência, 84-86
tipo abstrato de dados, 84-86
SET, 477-478
Shamir, 470-471
Shamos, 580-581, 672
Shiloach, 672
Shing, 287-288
símbolo de Jacobi, 462-463
símbolo de Legendre, 461-462
sistemas de encriptação, 467-468
skip list, 198-206
análise, 203-206
inserção, 201-203
níveis, 199-200
operações de atualização, 201-204
pesquisa, 200-202
remoção, 202-204
torres, 199-200
Sleator, 219-220, 672

sobreposição de subproblemas, 281-282
soma de prefixos, 47-48
soma geométrica, 675
soma telescópica, 49, 675
somatório, 35-36, 675
  geométrico, 35-36
splay tree, 187, 189-198
split, 170-171, 642-643
Stearns, 632
Steiglitz, 287-288, 632
Stephen, 445-446
STL, 145-146
stopwords, 428-429, 445-446
Strassen, 287-288
string
  nula, 417
  pesquisa de padrões, *veja* pesquisa de padrões
  prefixo, 417
  sufixo, 417
  tipo abstrato de dados, 417-418
string vazia, 417
subárvore, 86-87
subconjuntos seqüenciais, 648-649
subgrafo, 296
subgrafo de cobertura, 296
SUBGRAPH-ISOMORPHISM, 629-630
subseqüência, 438-439
SUBSET-SUM, 602-605, 619-620, 627-628, 632
substituição iterativa, 268-269
substituição local, 593-594
substring, 417
sufixo, 417
Szemerédi, 672

T

tabela apagável, 48
tabela de pesquisa, 149-153
tabela *hash*, 124-135
  agrupamento secundário, 131-132
  agrupamento, 131-132
  arranjo de *buckets*, 124-125
  capacidade, 124-125
  colisão, 124-125
  duplo *hash*, 131-132
  encadeamento, 129
  endereçamento aberto, 131-132
  hashing universal, 132-135
  resolução de colisões, 128-135
  teste linear, 131
  teste quadrático, 131-132
tamanho de entrada, 584
Tamassia, 145-146, 340-341, 502-503
Tarjan, 66-67, 145-146, 219-220, 261, 340-341, 379, 411-412, 672
Tel, 535-536
tempo de execução, 20-22, 25-28
tempo de início, 265-266
tempo pseudopolinomial, 284-285
teorema da Convolução, 485-486
teorema de Brent, 648-650
teorema de Cook-Levin, 590-591
teorema de Euler, 456-457
teorema do fluxo máximo e corte mínimo, 387-388
teorema fundamental da aritmética, 449
teoria de matróides, 287-288
teoria dos números, 449-466
Teseu, 292
teste da razão, 57-58
teste de potências, 58-59
teste de primalidade, 460-466
  algoritmo de Rabin-Miller, 464-465
  algoritmo de Solovay-Strassen, 461-464
teste linear, 131
teste quadrático, 131-132
thread, 75-77
tipo abstrato de dados
  árvore, 88-90
  conjunto, 229-239
  dicionário, 122-124, 148-149
  fila de prioridades, 104-105, 120-122
  fila, 73-74
  grafo, 293-299
  lista, 81-82
  partição, 231-239
  pilha, 70-71
  seqüência, 84-86
  string, 417-418
  vetor, 77-78
torre, 201-202
torre-de-dois, 234-235
trailer, 82
transferência, 172-174
transformada discreta de Fourier, 484-487
transformada rápida de Fourier, 481-499, 502-503
treap, 580-581
triangulação, 287-288
triangulação de Delaunay, 580-581
trie, 426-435
  comprimido, 429-430
  padrão, 426-427
troca de chave de Diffie e Hellman, 480-481
troca de chaves, 469
Tsakalidis, 219-220
TSP, 607-610, 613-614, 622, 630-632
Tukey, 502-503

U

Ullman, 145-146, 219-220, 261, 502-503, 672
último nodo, 108-109
underflow, 172-174, 642-643
união pelo tamanho, 233-234
union-find, 231-239
uso de espaço, 20

V

Valiant, 672
valor de *hash*, 126-127
valor esperado, 43-44
van Leeuwen, 340-341, 379, 411-412
variável aleatória, 43-44
varredura do plano, 556-561, 580-581
verificação, 587

vermelho duplo, 177-178
VERTEX-COVER 589-590, 599-604, 606, 612, 632
vértice, 293
  grau de entrada, 294
  grau de saída, 294
vértice de separação, 310-311
vértices finais, 294
vetor, 77-78-79-80, 123-124, 149-150
  tipo abstrato de dados, 77-78
vetor de *bits*, 256-257
vetor de distâncias, 524
Vishkin, 145-146, 672
Vismara, 580-581
Vitter, 66-67, 672
volta à direita, 564-565
Vuillemin, 580-581

W

Web *crawler*, 305-307, 438-439
Web *spider*, 305-307, 438-439
Wegner, 145-146
Williams, 145-146
Wood, 145-146, 580-581

X

X.509, 479-480

Y

Yannakakis, 632
Yao, 580-581
Yap, 502-503
Yellin, 145-146
Young, 632

Z

zigue, 190-191, 195-196
zigue-zague, 190-191, 195-196
zigue-zigue, 187, 189, 195-196